Oracle® OLAP

Java API Developer's Guide

11*g* Release 2 (11.2)

E10795-01

July 2009

Oracle OLAP Java API Developer's Guide, 11*g* Release 2 (11.2)

E10795-01

Primary Author: David McDermid

Contents

3 Discovering Metadata

4 Creating Metadata and Analytic Workspaces

5 Understanding Source Objects

6 Making Queries Using Source Methods

7 Using a TransactionProvider

8 Understanding Cursor Classes and Concepts

9 Retrieving Query Results

10 Creating Dynamic Queries

A Setting Up the Development Environment

B SingleSelectionTemplate Class

Index

List of Figures

List of Examples

Preface

Oracle OLAP Java API Developer's Guide introduces Java programmers to the Oracle OLAP Java API, which is the Java application programming interface for Oracle OLAP. Through Oracle OLAP, the OLAP Java API provides access to data stored in an Oracle database. The OLAP Java API capabilities for creating and maintaining analytic workspaces, and for querying, manipulating, and presenting data are particularly suited to applications that perform online analytical processing (OLAP) operations.

The preface contains these topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

Oracle OLAP Java API Developer's Guide is intended for Java programmers who are responsible for creating applications that do one or more of the following:

- Implement an Oracle OLAP metadata model.
- Define, build, and maintain analytic workspaces.
- Perform analysis using Oracle OLAP.

To use this manual, you should be familiar with Java, relational database management systems, data warehousing, OLAP concepts, and Oracle OLAP.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Related Documents

For more information, see these Oracle resources:

- *Oracle OLAP Java API Reference*
- *Oracle OLAP User's Guide*
- *Oracle OLAP DML Reference*
- *Oracle Warehouse Builder Concepts*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New

This preface describes the features of the Oracle OLAP Java API that are new in the 11*g* releases of Oracle OLAP.

What's New in 11.2

This section describes the features of the Oracle OLAP Java API that Oracle has added since the initial 11*g* Release 1 (11.1). For a complete list of the packages, classes, fields, and methods that are new since 11.1, see the Overview of *Oracle OLAP Java API Reference*.

The following topics describe the new features since 11.1.

- Supporting Legacy Metadata Objects
- Populating Hierarchy Lineage in an OLAP View
- Supporting Materialized Views
- Enhancing the Build Process
- Aggregating over Specified Dimension Members
- Specifying a Language for a Dimension Attribute
- Classifying and Grouping Metadata
- Controlling the Export of XML Attributes

Supporting Legacy Metadata Objects

To support legacy 10*g* metadata objects, the Oracle OLAP Java API includes the following functionality.

- Allowing 10*g* and 11*g* metadata objects in the same session. The `oracle.olapi.data.source.DataProvider` class has the new `ALL` metadata reader mode setting. For more information, see "Describing Namespaces" on page 2-4.
- Attributing namespaces. To allow a 10*g* metadata object to exist in the same session as 11*g* objects, Oracle OLAP assigns a namespace to the 10g object. The namespace identifies the metadata format and the type of object. For more information on namespaces, see "Describing Namespaces" on page 2-4.
- Renaming persistent metadata objects. Previously, you could use the `MdmObject.setName` method to specify a name for a persistent metadata object only once. You could not change the name after setting it. Now, you can use the `setName` method to rename a metadata object. For more information on setting names, see "Getting and Setting Names" on page 2-3.

Populating Hierarchy Lineage in an OLAP View

In a column for an attribute in an OLAP view, you can now automatically populate the rows for lower levels in a dimension hierarchy with the attribute values that are mapped at a higher level. For more information, see "Populating OLAP Views with Hierarchical Attribute Values" on page 2-23.

Supporting Materialized Views

The following new features affect materialized views for OLAP cubes.

- Specifying attribute column names for materialized views. With the `setETAttrPrefix` method of the `MdmDimensionality` object for a dimension of an OLAP cube, you can specify the prefix that Oracle OLAP uses in naming the columns for the attributes of the dimension in the materialized view for the cube. For more information, see "Preparing Attributes for Materialized Views" on page 2-25.
- Including populated lineage attributes in a materialized view that is available to the query rewrite system. If the materialized view has a column for the same attribute at different levels of a hierarchy, you can specify the new `REWRITE_WITH_ATTRIBUTES_MV_OPTION` materialized view option of the `AWCubeOrganization` class. That option populates the column for an attribute for a level in the view with the values of the same attribute for lower levels. For more information, see "Representing Cubes" on page 2-14.

Enhancing the Build Process

The following topics describe new features that affect building analytic workspace metadata objects.

- Specifying Serial or Parallel Builds
- Tracking the Progress of a Build

Specifying Serial or Parallel Builds

You can now create an `oracle.olapi.syntax.LoadCommand` that specifies the use of serial or parallel processing when loading data in an analytic workspace. The `LoadCommand` also has a `PRUNE` option, which specifies that the build spawns jobs only for the partitions for which measure data exists.

The `LoadCommand` class now has constructor methods as well as static constant fields that produce `LoadCommand` objects. One of the constructors takes the name of a `CubeMap`. With that constructor, you can specify loading data from single `CubeMap` rather than from all `CubeMap` objects.

The `ClearCommand` and `SolveCommand` classes also have serial and parallel processing options.

Tracking the Progress of a Build

With an `oracle.olapi.data.source.BuildResult`, an application can get the identification number for a build process. The `DataProvider` class has `executeBuild` methods that take a `BuildResult`.

An analytic workspace automatically generates the identification number during a build. You can use the build number to track the progress of a build.

Aggregating over Specified Dimension Members

With the `oracle.olapi.syntax.AggregationFunctionExpression` class, in an `MdmModel` you can aggregate measure values over specified dimension members.

You specify the dimension members with an AggregateOverMembersClause. You can use an AggregationFunctionExpression as the Expression for an MdmCustomMember or the member expression of an MdmAssignment in the MdmModel.

Specifying a Language for a Dimension Attribute

You can now specify a language when mapping an attribute and you can map multiple languages to the same attribute. For more information, see "Specifying a Language for an Attribute" on page 2-23 and "Specifying Multilingual Attributes" on page 2-23.

Classifying and Grouping Metadata

The following new features allow you to add identifying metadata to objects.

- Classifying objects. With the addObjectClassification method of an MdmObject, you can add metadata to that object. For more information, see "Using Classifications" on page 2-7.
- Grouping attributes. With the setAttributeGroupName method of an MdmBaseAttribute, you can specify a name for an attribute group. For more information, see "Grouping Attributes" on page 2-23.

Controlling the Export of XML Attributes

An implementation of the oracle.olapi.metadata.XMLWriterCallback interface provides Oracle OLAP the means to call back to an application while the MdmMetadataProvider is exporting the XML definition of an object. With an XMLWriterCallback, the application can specify whether or not to exclude an attribute or an owner name from the exported XML. For more information see "Exporting and Importing Metadata as XML Templates" on page 2-8.

What's New in 11.1

Some aspects of the Oracle OLAP Java API are much the same as in previous releases, such as the ability to create queries with classes in the oracle.olapi.data.source package and to retrieve the data with classes in the oracle.olapi.data.cursor package. However, in Oracle OLAP 11g Release 1 (11.1) the metadata model of the API has changed and has many new features. The major new features are presented in the following topics.

- Create Persistent Metadata Objects
- Restrict Access to Persistent Objects
- Define and Build Analytic Workspaces
- Export and Import XML Definitions
- Use SQL-Like Expression Syntax
- Share a Connection Between Multiple Sessions
- Specify a Metadata Reader Mode

Create Persistent Metadata Objects

The Oracle OLAP Java API now has the ability to create and maintain persistent metadata objects. The Oracle Database stores the metadata objects in the Oracle data dictionary.

To provide this new functionality, the Oracle OLAP Java API substantially revises the metadata model. The new model includes several new packages and has significant changes to some existing packages. For example, the `oracle.olapi.metadata.mdm` package has many new classes. It also has many new methods added to existing classes. For information on OLAP metadata objects, see Chapter 2, "Understanding OLAP Java API Metadata".

Some classes and methods are deprecated in the new model. For example, all of the classes in the `oracle.olapi.metadata.mtm` package are deprecated, and methods of other classes that use the `mtm` classes are also deprecated. Some `mtm` classes mapped transient `mdm` objects to relational database structures, such as columns in tables and views. Other `mtm` classes specified how Oracle OLAP performed operations such as aggregation or allocation of the values of custom measures. That functionality is replaced by classes in the `oracle.olapi.metadata` subpackages `deployment`, `mapping`, and `mdm`, and the `oracle.olapi.syntax` package. With the new classes, an application can create permanent metadata objects, map them to data sources, and specify the operations that provide values for measures.

Restrict Access to Persistent Objects

When an application commits the `Transaction` in which it has created top-level objects in the OLAP metadata model, such as instances of classes like `AW`, `MdmCube`, and `MdmPrimaryDimension`, those objects then exist in the Oracle data dictionary. They are available for use by ordinary SQL queries as well as for use by applications that use the Oracle OLAP Java API.

Because the metadata objects exist in the Oracle data dictionary, an Oracle Database DBA can restrict access to certain types of the metadata objects. A client application can set such restrictions by using the JDBC API to send standard SQL `GRANT` and `REVOKE` commands through the JDBC connection for the user session.

Define and Build Analytic Workspaces

An application can now define, build, and maintain analytic workspaces. This new functionality is provided by classes in the `oracle.olapi.metadata` subpackages `deployment`, `mapping`, and `mdm`, and the `oracle.olapi.syntax package`. In 10g releases of Oracle Database, that functionality was provided by a separate API, the Oracle OLAP Analytic Workspace Java API, which is entirely deprecated in this release. For more information see Chapter 2.

Export and Import XML Definitions

After defining a metadata object, an application can export that definition in an XML format. Analytic Workspace Manager refers to such a saved definition as a template. An application can also import the XML definition of a metadata object. The `MdmMetadataProvider` class has methods for exporting and importing the XML. For more information see "Exporting and Importing Metadata as XML Templates" on page 2-8.

Use SQL-Like Expression Syntax

With the classes in the `oracle.olap.syntax package`, an application can create Java objects that are based on SQL-like expressions, functions, operators, and conditions. The `SyntaxObject` class has `fromSyntax` and `toSyntax` methods that an application can use to convert SQL expressions into Java objects or to get the SQL syntax from a Java object.

An application can create an `Expression` object by using the `SyntaxObject.fromSyntax` method or by using a constructor. For example, the following code creates a StringExpression using a fromSyntax method and another StringExpression using a constructor method. The mp object is the MdmMetadataProvider for the session.

```
StringExpression strExp1 = (StringExpression)
                     SyntaxObject.fromSyntax("'Hello world from syntax.'", mp);
StringExpression strExp2 = new StringExpression("Hello world from constructor.");
```

Share a Connection Between Multiple Sessions

Another new feature is the ability to have multiple user sessions that share the same JDBC connection to the Oracle Database instance and that share the same cache of metadata objects. This ability is provided by the UserSession class in the oracle.olapi.session package.

Specify a Metadata Reader Mode

To support legacy applications, the OLAP Java API provides a means of specifying a metadata reader that can recognize metadata objects that were created by a previous method. For more information, see "Supporting Legacy Metadata Objects" on page 2-4.

1

Introduction to the OLAP Java API

This chapter introduces the Oracle OLAP Java application programming interface (API). The chapter includes the following topics:

- OLAP Java API Overview
- Accessing Data Through the OLAP Java API
- Sample Schema for OLAP Java API Examples
- Tasks That an OLAP Java API Application Performs

OLAP Java API Overview

The Oracle OLAP Java API is an application programming interface that provides access to the online analytic processing (OLAP) technology in Oracle Database with the OLAP option. This section lists operations that an OLAP Java API client application can perform, describes the classes in the OLAP Java API, describes the objects in a dimensional data model, and discusses organizing data for online analytical processing.

For a description of the advantages of OLAP technology, see *Oracle OLAP User's Guide*. That document describes the capabilities that Oracle OLAP provides for the analysis of multidimensional data by business intelligence and advanced analytical applications. It describes in depth the dimensional data model, and it discusses the database administration and management tasks related to Oracle OLAP.

What the OLAP Java API Can Do

Using the OLAP Java API, your can develop client applications that do the following operations.

- Establish one or more user sessions in a JDBC connection to an Oracle Database instance. Multiple user sessions can share the same connection and the same cache of metadata objects.
- Manage OLAP transactions with the database.
- Create and maintain analytic workspaces.
- Implement a dimensional data model using OLAP metadata objects.
- Create logical metadata objects and map them to relational sources.
- Deploy the metadata objects as an analytic workspace or as relational tables and views and commit the objects to the database.

- Explore the metadata to discover the data that is available for viewing or for analysis.
- Construct analytical queries of the multidimensional data. Enable end users to create queries that specify and manipulate the data according to the needs of the user (for example, selecting, aggregating, and calculating data).
- Modify queries, rather than totally redefine them, as application users refine their analyses.
- Retrieve query results that are structured for display in a multidimensional format.

For more information on some of these operations, see "Tasks That an OLAP Java API Application Performs".

Describing the Classes in the OLAP Java API

The OLAP Java API has classes that represent the following types of objects.

- User sessions
- Transactions
- Metadata objects
- Build items, processes, specifications, and commands
- Queries
- Cursors that retrieve the data of a query
- Expressions that specify data objects, such as a column in a relational table or view, or that specify a function or command that operates on data

Table 1–1 lists packages that contain the majority of the OLAP Java API classes. These packages are under the `oracle.olapi` package. The table contains brief descriptions of the package contents.

Table 1–1 Packages of the OLAP Java API under oracle.olapi

Package	Description
`data.cursor`	Contains classes that represent cursor managers and cursors that retrieve the data specified by a `Source` object. For information on `Cursor` objects, see Chapter 8, "Understanding Cursor Classes and Concepts" and Chapter 9, "Retrieving Query Results".
`data.source`	Contains classes that represent data sources and cursor specifications. You use `Source` objects to create queries of the data store. With the `Template` class you can incrementally build a `Source` object that represents a query that you can dynamically modify. For information on `Source` objects, see Chapter 5, "Understanding Source Objects" and Chapter 6, "Making Queries Using Source Methods". For information on `Template` objects, see Chapter 10, "Creating Dynamic Queries".

`metadata`
`metadata.deployment`
`metadata.mapping`
`metadata.mdm`
	Contains classes that represent metadata objects, classes that map the metadata objects to relational data sources, and classes that deploy the metadata objects in an analytic workspace or in relational database structures. For a description of these packages, see Chapter 2, "Understanding OLAP Java API Metadata". For information on using the classes in these packages, see Chapter 3, "Discovering Metadata" and Chapter 4, "Creating Metadata and Analytic Workspaces".

Table 1–1 (Cont.) Packages of the OLAP Java API under oracle.olapi

Package	Description
`resource`	Contains classes that support the internationalization of messages for `Exception` classes.
`session`	Contains a class that represents a session in a connection to an Oracle database.
`syntax`	Contains classes that represent the items and commands that specify how Oracle OLAP builds analytic workspace objects and classes that implement a syntax for creating SQL-like expressions. You use `Expression` objects in mapping metadata objects to relational data sources such as columns in a table or a view. You also use `Expression` objects to specify calculations and analytical operations for some metadata objects.
`transaction`	Contains classes that represent transactions with Oracle OLAP in an Oracle Database instance. You use `Transaction` objects to commit changes to the database. For information on `Transaction` objects, see Chapter 7, "Using a TransactionProvider".

The OLAP Java API also has packages organized under the `oracle.express` package. These packages date from the earliest versions of the API. The classes that remain in these packages are mostly `Exception` classes for exceptions that occur during interactions between Oracle OLAP and a client application.

For information on obtaining the OLAP Java API software and on the requirements for using it to develop applications, see Appendix A, "Setting Up the Development Environment."

Describing the Dimensional Data Model

Data warehousing and OLAP applications are based on a multidimensional view of data. This view is implemented in a dimensional data model that includes the following dimensional objects. For more detailed information about all of these concepts, see *Oracle OLAP User's Guide* and *Oracle Warehouse Builder Concepts*.

Cubes

Cubes are containers for measures that have the same set of dimensions. A cube usually corresponds to a single relational fact table or view. The measures of a cube contain facts and the dimensions give shape to the fact data. Typically, the dimensions form the edges of the cube and the measure data is the body of the cube. For example, you could organize data on product units sold into a cube whose edges contain values for members from time, product, customer, and channel dimensions and whose body contains values from a measure of the quantity of units sold and a measure of sales amounts.

The OLAP concept of an cube edge is not represented by a metadata object in the OLAP Java API, but edges are often incorporated into the design of applications that use the OLAP Java API. Each edge contains values of members from one or more dimensions. Although there is no limit to the number of edges on a cube, data is often organized for display purposes along three edges, which are referred to as the row edge, column edge, and page edge.

Measures
Measures contain fact data in a cube. The measure values are organized and identified by dimensions. Measures are usually multidimensional. Each measure value is identified by a unique set of dimension members. This set of dimension members is called a *tuple*.

Dimensions
Dimensions contain lists of unique values that identify and categorize data in a measure. Commonly-used dimensions are customers, products, and times. Typically, a dimension has one or more hierarchies that organize the dimension members into parent-child relationships. A hierarchy has one or more levels.

By specifying dimension members, measures, and calculations to perform on the data, end users formulate business questions and get answers to their queries. For example, using a time dimension that categorizes data by month, a product dimension that categorizes data by unit item, and a measure that contains data for the quantities of product units sold by month, you can formulate a query that asks if sales of a product unit were higher in January or in June.

Hierarchies
Hierarchies are components of a dimension that organize dimension members into parent-child relationships. Typically, in the user interface of a client application, an end user can expand or collapse a hierarchy by drilling down or up among the parents and children. The measure values for the parent dimension members are aggregations of the values of the children.

A dimension can have more than one hierarchy. For example, a time dimension could have a calendar year hierarchy and a fiscal year hierarchy. A hierarchy can be level-based or value-based.

In a level-based hierarchy, a parent must be in a higher level than the children of that parent. In a cube, the measure values for the parents are typically aggregated from the values of the children. For example, a time dimension might have levels for year, quarter, and month. The month level contains the base data, which is the most detailed data. The measure value for a quarter is an aggregation of the values of the months that are the children of the quarter and the measure value for a year is the aggregation of the quarters that are children of the year. Typically each level is mapped to a different column in the relational dimension table.

In a value-based hierarchy, the parent and the child dimension members typically come from the same column in the relational table. Another column identifies the parent of a member. For example, a value hierarchy could contain all employees of a company and identify the manager for each employee that has one. All employees, including managers, would come from the same column. Another column would contain the parent of a member.

Levels
Levels are components of a level-based hierarchy. A level can be associated with more than one hierarchy. A dimension member can belong to only one level.

A level typically corresponds to a column in a dimension table or view. The base level is the primary key.

Attributes
Attributes contain information related to the members of a dimension. An end user can use an attribute to select data. For example, an end user might select a set of products using an attribute that has a descriptive name of each product. An attribute is contained by a dimension.

Queries

A query is a specification for a particular set of data. The term *query* in the OLAP Java API refers to a `Source` object that specifies a set of data and can include aggregations, calculations, or other operations to perform using the data. The data and the operations on it define the result set of the query. In this documentation, the general term *query* refers to a `Source` object.

The API has a `Query` class in the `oracle.olapi.syntax` package. A `Query` represents a multirow, multicolumn result set that is similar to a relational table, a SQL `SELECT` statement, or an OLAP function. You use a `Query` object in mapping a dimension or measure to a relational table or view.

Implementing the Dimensional Data Model

In the OLAP Java API, the dimensional data objects are represented by Multidimensional Model (MDM) classes. These classes are in the `oracle.olapi.metadata.mdm` package. Related classes are in the `oracle.olapi.metadata` package and the other packages under it. For detailed information about those classes, see Chapter 2, "Understanding OLAP Java API Metadata".

Organizing the Data for OLAP

The OLAP Java API makes it possible for Java applications (including applets) to access data that resides in an Oracle data warehouse. A data warehouse is a relational database that is designed for query and analysis, rather than for transaction processing. Warehouse data often conforms to a star schema, which is a dimensional data model for a relational database. A star schema consists of one or more fact tables and one or more dimension tables. The fact tables have columns that contain foreign keys to the dimension tables. Typically, a data warehouse is created from a transaction processing database by an extraction transformation transport (ETT) tool, such as Oracle Warehouse Builder.

For the data in a data warehouse to be accessible to an OLAP Java API application, a database administrator must ensure that the data warehouse is configured according to an organization that is supported by Oracle OLAP. The star schema is one such organization, but not the only one. See *Oracle OLAP User's Guide* for information about supported data warehouse configurations.

Once the data is organized in the warehouse, you can use an OLAP Java API application to design an OLAP dimensional data model of cubes, measures, dimensions, and so on, and to create the logical OLAP metadata objects that implement the model. You map the metadata objects to data in the warehouse and build an analytic workspace. Building the analytic workspace populates the OLAP views and other storage structures with the data that the OLAP metadata objects represent.

You can also use Analytic Workspace Manager to do the same tasks. See *Oracle OLAP User's Guide* for information about creating an analytic workspace with Analytic Workspace Manager.

An OLAP Java API application can get the OLAP metadata objects created either by Analytic Workspace Manager or through the OLAP Java API. It can use the metadata objects to create queries that operate on the data in the warehouse.

The collection of warehouse data in an analytic workspace is the data store to which the OLAP Java API gives access. Of course, the scope of the data that a user has access to is limited by the privileges granted to the user by the database administrator.

In addition to ensuring that data and metadata have been prepared appropriately, you must ensure that application users can make a JDBC connection to the data store and that users have database privileges that give them access to the data. For information about specifying privileges, see *Oracle OLAP User's Guide*. For information about establishing a connection, see Chapter 3, "Discovering Metadata".

Accessing Data Through the OLAP Java API

Oracle OLAP metadata objects organize and describe the data that is available to a client application. The metadata objects contain other information, as well, such as the data type of the data. However, you cannot retrieve data directly from a metadata object. To specify the data that you want, you must create a query. In specifying the data, you usually must specify one or more dimension member values. To retrieve the specified data, you create a `Cursor`. This section briefly describes those actions.

Another way that you can query the data contained in OLAP metadata objects is through SQL queries of the views that Oracle OLAP creates for the metadata objects. For information about querying these views, see "Using OLAP Views" in Chapter 2, "Understanding OLAP Java API Metadata".

Creating Queries

Queries are represented by `oracle.olapi.data.source.Source` objects. You get a `Source` from the metadata object and use that `Source` object in specifying the data that you want to get. `Source` classes have methods for selecting and performing operations on the data. You can use the methods to manipulate data in any way that the user requires. For information about `Source` objects, see Chapter 5, "Understanding Source Objects" and Chapter 6, "Making Queries Using Source Methods".

Specifying Dimension Members

The members of an Oracle OLAP dimension are usually organized into one or more hierarchies. Some hierarchies have parent-child relationships based on levels and some have those relationships based on values. The value of each dimension member must be unique.

The OLAP Java API uses a three-part format to uniquely identify a dimension member. The format contains the hierarchy, the level, and the value of the dimension member, and thereby identifies a unique value in the dimension. The first part of a unique value is the name of the hierarchy object, the second part is the name of the level object, and the third part is the value of the member in the level. The parts of the unique value are separated by a value separation string, which by default is double colons (`::`). The following is an example of a unique member value of a level named `YEAR` in a hierarchy named `CALENDAR_YEAR` in a dimension named `TIME_AWJ`.

```
CALENDAR_YEAR::YEAR::CY2001
```

The third part of a unique value is the local value. The local value in the preceding example identifies the calendar year 2001.

Creating Cursors

To retrieve the data specified by a `Source`, you create an `oracle.olapi.data.cursor.Cursor` for that `Source`. You then use this `Cursor` to request and retrieve the data from the data store. You can specify the amount of data that the `Cursor` retrieves in each fetch operation (for example, enough to fill a 40-cell table in the user interface). Oracle OLAP then efficiently manages the timing, sizing, and caching of the data blocks that it retrieves for your application, so that you do not need to do so. For information about Cursor objects, see Chapter 8, "Understanding Cursor Classes and Concepts" and Chapter 9, "Retrieving Query Results".

Sample Schema for OLAP Java API Examples

The examples of OLAP Java API code in this documentation are excerpts from a set of example programs that are available on the Oracle Technology Network (OTN) Web site. One example creates and builds an analytic workspace named GLOBAL_AWJ. Other examples query that analytic workspace. The metadata objects in the analytic workspace are mapped to columns in relational tables that are in the Global schema.

From the OTN Web site, you can download a file that contains SQL scripts that create the Global schema and a file that contains the example programs. The OTN Web site is at

http://www.oracle.com/technology/products/bi/olap/index.html

To get either file, select **Sample Code and Schemas** in the Download section of the Web page. To get the sample schema, select **Global Schema 11g**. To get the example programs, select **Example Programs for Documentation** and then select Download the **Example Programs for 11g Release 2 (11.2)** to download the compressed file examples.zip.

The example programs are in a package structure that you can easily add to your development environment. The classes include a base class that the example program classes extend, and utility classes that they use. The base class is BaseExample11g.java. The utility classes include Context11g.java and CursorPrintWriter.java. The Context11g.java class has methods that create a connection to an Oracle Database instance, that store metadata objects, that return the stored metadata objects, and that create Cursor objects. The CursorPrintWriter.java class is a PrintWriter that has methods that display the contents of Cursor objects.

The OLAP metadata objects are created and built by the BuildAW11g.java program. Those metadata objects include the following:

- GLOBAL_AWJ, which is the analytic workspace that contains the other objects.
- PRODUCT_AWJ, which is a dimension for products. It has one hierarchy named PRODUCT_PRIMARY. The lowest level of the hierarchy has product item identifiers and the higher levels have product family, class, and total products identifiers.
- CUSTOMER_AWJ, which is a dimension for customers. It has two hierarchies named SHIPMENTS and MARKETS. The lowest level of each hierarchy has customer identifiers and higher levels have warehouse, region, and total customers, and account, market segment, and total market identifiers, respectively.
- TIME_AWJ, which is a dimension for time values. It has a hierarchy named CALENDAR_YEAR. The lowest level has month identifiers, and the other levels have quarter and year identifiers.
- CHANNEL_AWJ, which is a dimension for sales channels. It has one hierarchy named CHANNEL_PRIMARY. The lowest level has sales channel identifiers and the higher level has the total channel identifier.
- UNITS_CUBE_AWJ, which is a cube that contains the measures UNITS and SALES. UNITS has values for the quantities of product units sold. SALES has the dollar amounts for the sales of product units. The cube is dimensioned by all four

dimensions. The aggregation method for the cube is SUM, in which each the value for each parent is the sum of the values of the children of the parent.

- PRICE_CUBE_AWJ, which is a cube that contains the measures UNIT_COST and UNIT_PRICE. UNIT_COST has the costs of the units. UNIT_PRICE has the prices of the units. The cube is dimensioned by the PRODUCT_AWJ and TIME_AWJ dimensions. The aggregation method for the cube is AVG, in which the value for each parent is the average of the values of the children of the parent.

For an example of a program that discovers the OLAP metadata for the analytic workspace, see Chapter 3, "Discovering Metadata".

Tasks That an OLAP Java API Application Performs

A client application that uses the OLAP Java API typically performs the following tasks:

1. Connects to the data store and creates a DataProvider and a UserSession
2. Creates or discovers metadata objects
3. Deploys, maps, and builds metadata objects, as needed
4. Specifies queries that select and manipulate data
5. Retrieves query results

The rest of this topic briefly describes these tasks, and the rest of this guide provides detailed information about how to accomplish them.

Task 1: Connect to the Data Store and Create a DataProvider and UserSession

You connect to the data store by identifying some information about the target Oracle Database instance and specifying this information in a JDBC connection method. Having established a connection, you create a DataProvider and use it and the connection to create a UserSession. For more information about connecting and creating a DataProvider and UserSession, see "Connecting to Oracle OLAP" in Chapter 3.

Task 2: Create or Discover Metadata Objects

You use the DataProvider to get an MdmMetadataProvider. The MdmMetadataProvider gives access to all of the metadata objects in the data store. You next obtain the MdmRootSchema object by calling the getRootSchema method of the MdmMetdataProvider. The MdmRootSchema object contains all of the OLAP metadata objects in the database. From the MdmRootSchema, you get the MdmDatabaseSchema objects for the schemas that the current user has permission to access. An MdmDatabaseSchema represents a named Oracle Database user as returned by the SQL statement SELECT username FROM all_users.

From an MdmDatabaseSchema, you can discover the existing metadata objects that are owned by the schema or you can create new ones. Methods such as getMeasures and getDimensions get all of the measures or dimensions owned by the MdmDatabaseSchema. Methods such as findOrCreateAW and findOrCreateCube get an analytic workspace or cube, if it exists, or create one if it does not already exist.

From a top-level metadata object contained by the MdmDatabaseSchema, such as an analytic workspace, cube, or dimension, you can get the objects that it contains. For example, from an MdmPrimaryDimension, you can get the hierarchies, levels, and attributes that are associated with it. Having determined the metadata objects that are available to the user, you can present relevant lists of objects to the user for data selection and manipulation.

For a description of the metadata objects, see Chapter 2, "Understanding OLAP Java API Metadata". For information about how you can discover the available metadata, see Chapter 3, "Discovering Metadata".

Task 3: Deploy, Map, and Build Objects

If you create a new `MdmCube` or `MdmPrimaryDimension`, you must deploy it as an analytic workspace object or as a relational OLAP (Rolap) object. To deploy a cube, you call an `MdmCube` method such as `findOrCreateAWCubeOrganization`. To deploy a `dimension`, you call an `MdmPrimaryDimension` method such as `findOrCreateAWPrimaryDimensionOrganization`.

If you create a new metadata object that represents data, you must specify an `Expression` that maps the metadata object to a relational source table or view, or that Oracle OLAP uses to generate the data. For objects that are contained by an analytic workspace, you can build the metadata objects after mapping them. For information on creating metadata, deploying, mapping, and building metadata objects, see Chapter 4, "Creating Metadata and Analytic Workspaces".

Task 4: Select and Calculate Data Through Queries

An OLAP Java API application can construct queries against the data store. A typical application user interface provides ways for the user to select data and to specify the operations to perform using the data. Then, the data manipulation code translates these instructions into queries against the data store. The queries can be as simple as a selection of dimension members, or they can be complex, including several aggregations and calculations involving the measure values that are specified by selections of dimension members.

The OLAP Java API object that represents a query is a `Source`. Metadata objects that represent data are extensions of the `MdmSource` class. From an `MdmSource`, such as an `MdmMeasure` or an `MdmPrimaryDimension`, you can get a `Source` object. With the methods of a `Source` object, you can produce other `Source` objects that specify a selection of the elements of the `Source`, or that specify calculations or other operations to perform on the values of a `Source`.

If you are implementing a simple user interface, then you might use only the methods of a `Source` object to select and manipulate the data that users specify in the interface. However, if you want to offer your users multistep selection procedures and the ability to modify queries or undo individual steps in their selections, then you should design and implement `Template` classes. Within the code for each `Template`, you use the methods of the `Source` classes, but the `Template` classes themselves allow you to dynamically modify and refine even the most complex query. In addition, you can write general-purpose `Template` classes and reuse them in various parts of your application.

For information about working with `Source` objects, see Chapter 5, "Understanding Source Objects". For information about working with `Template` objects, see Chapter 10, "Creating Dynamic Queries".

Task 5: Retrieve Query Results

When users of an OLAP Java API application are selecting, calculating, combining, and generally manipulating data, they also want to see the results of their work. This means that the application must retrieve the result sets of queries from the data store and display the data in multidimensional form. To retrieve a result set for a query through the OLAP Java API, you create a `Cursor` for the `Source` that specifies the query.

You can also get the SQL that Oracle OLAP generates for a query. To do so, you create a `SQLCursorManager` for the `Source` instead of creating a `Cursor`. The `generateSQL` method of the `SQLCursorManager` returns the SQL specified by the `Source`. You can then retrieve the data by means outside of the OLAP Java API.

Because the OLAP Java API was designed to deal with a multidimensional view of data, a `Source` can have a multidimensional result set. For example, a `Source` can represent an `MdmMeasure` that is dimensioned by four `MdmPrimaryDimension` objects. Each `MdmPrimaryDimension` has an associated `Source`. You can create a query by joining the `Source` objects for the dimensions to the `Source` for the measure. The resulting query has the `Source` for the measure as the base and it has the `Source` objects for the dimensions as outputs.

A `Cursor` for a query `Source` has the same structure as the `Source`. For example, the `Cursor` for the `Source` just mentioned has base values that are the measure data. The `Cursor` also has four outputs. The values of the outputs are those of the `Source` objects for the dimensions.

To retrieve all of the items of data through a `Cursor`, you can loop through the multidimensional `Cursor` structure. This design is well adapted to the requirements of standard user interface objects for painting the computer screen. It is especially well adapted to the display of data in multidimensional format.

For more information about using `Source` objects to specify a query, see Chapter 5, "Understanding Source Objects". For more information about using `Cursor` objects to retrieve data, see Chapter 8, "Understanding Cursor Classes and Concepts". For more information about the `SQLCursorManager` class, see *Oracle OLAP Java API Reference*.

2

Understanding OLAP Java API Metadata

This chapter describes the classes in the Oracle OLAP Java API that represent OLAP dimensional and relational metadata objects. It also describes the classes that provide access to the metadata objects and to data sources, or that contain information about the metadata objects. This chapter includes the following topics:

- Overview of OLAP Java API Metadata Classes
- Identifying, Describing, and Classifying Metadata Objects
- Providing Metadata Objects
- Providing Access to Data Sources

For more information on getting existing metadata objects, see Chapter 3, "Discovering Metadata". For more information on creating metadata objects, see Chapter 4, "Creating Metadata and Analytic Workspaces".

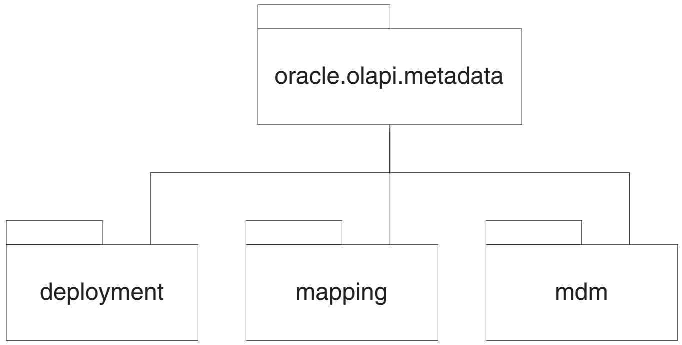
Overview of OLAP Java API Metadata Classes

Chapter 1 describes the OLAP dimensional data model and briefly mentions some of the OLAP Java API classes that implement that model. Those classes are in the `oracle.olapi.metadata` packages. Using those classes, you can do the following tasks.

- Gain access to the available metadata objects
- Create new metadata objects
- Deploy metadata objects in an analytic workspace or as relational objects
- Map metadata objects to data sources
- Export metadata objects to XML or import them from XML
- Create `Source` objects to query the data

Figure 2–1 shows the `oracle.olapi.metadata` packages.

Figure 2–1 The oracle.olapi.metadata Packages



The packages are the following:

- `oracle.olapi.metadata`, which has interfaces and abstract classes that specify the most basic characteristics of metadata objects and metadata providers.
- `oracle.olapi.metadata.mdm`, which has classes that implement the MDM (multidimensional model) metadata model. This package has classes that represent the metadata objects, classes that provide access to those objects, and classes that contain descriptive information about the objects.
- `oracle.olapi.metadata.deployment`, which has classes that specify the organization of a metadata object as an analytic workspace object or as a relational object.
- `oracle.olapi.metadata.mapping`, which has classes that map a metadata object to relational data sources.

Some of the classes in the `oracle.olapi.metadata.mdm` package directly correspond to OLAP dimensional metadata objects. Table 2–1 presents some of these correspondences.

Table 2–1 Corresponding Dimensional and MDM Objects

Dimensional Metadata Objects	MDM Metadata Objects
Cube	`MdmCube`
Measure	`MdmBaseMeasure`
Calculated measure	`MdmDerivedMeasure`
Measure folder	`MdmOrganizationalSchema`
Dimension	`MdmTimeDimension` and `MdmStandardDimension`
Hierarchy	`MdmLevelHierarchy` and `MdmValueHierarchy`
Level	`MdmDimensionLevel` and `MdmHierarchyLevel`
Attribute	`MdmBaseAttribute` and `MdmDerivedAttribute`

Other classes in the package correspond to relational objects. Table 2–2 shows those correspondences.

Table 2–2 Corresponding Relational and MDM Objects

Relational Objects	MDM Metadata Objects
Schema	`MdmDatabaseSchema`
Table	`MdmTable`
Table column	`MdmColumn`

Identifying, Describing, and Classifying Metadata Objects

Most OLAP Java API metadata objects have a unique identifier (ID), a name, and an owner or a containing object. You can also associate descriptions and classifications to most metadata objects.

Most metadata classes extend the abstract `oracle.olapi.metadata.BaseMetadataObject` class. A `BaseMetadataObject` can have a name and an ID. You can get most metadata objects by name. The ID is used internally by Oracle OLAP, but an application can also use the ID to get some metadata objects.

A `BaseMetadataObject` also has an owner, which is returned by the `getOwner` method. For most metadata objects, the owner is an `MdmDatabaseSchema`. For the `MdmRootSchema` and `MdmMeasureDimension` objects, the owner is the root schema. For an `MdmViewColumn`, which is not a subclass of `BaseMetadataObject`, the `getOwner` method returns the owning implementation of the `MdmViewColumnOwner` interface, such as an `MdmPrimaryDimension`, an `MdmBaseAttribute`, or an `MdmMeasure`. An `MdmViewColumn` represents a column in an OLAP view. For information on OLAP views, see "Using OLAP Views" on page 2-26.

Some `BaseMetadataObject` objects are contained by the metadata object that created them. For example, an `MdmBaseMeasure` is contained by the `MdmCube` that created it. You can get the container for a metadata object by calling the `getContainedByObject` method.

The `MdmObject` class, which is an abstract subclass of `BaseMetadataObject`, adds associations with descriptive objects and classifications. Typically, a descriptive object contains a name or descriptive text that you associate with the metadata object itself. Applications often use a descriptive object for display purposes in a user interface. A classification is a string value that your application assigns to the metadata object. Your application handles the classification for whatever purpose you want.

Identifying Objects

You can identify a `BaseMetadataObject` object by name and by ID. Namespaces identify the type and the format of legacy metadata objects.

Getting and Setting Names

Most metadata objects have a name that you can get by calling the `getName` method of the object. For some objects, you can assign a name when you create the object. For example, an `oracle.olapi.metadata.deployment.AW` object represents an analytic workspace. When you create an `AW` by calling the `findOrCreateAW` method of an `MdmDatabaseSchema`, you use the `publicName` parameter of the method to specify a name for the `AW` object that the method returns.

For some objects, you can use the setName method to change the name of an existing object. For example, you can change the name of an MdmStandardDimension by calling the setName method of the dimension object. The new name does not take effect until you commit the root transaction of the session. After you call setName, but before you commit the root transaction, the getNewName method returns the new name while the getName method returns the existing name.

You can get some objects by name from an MdmDatabaseSchema. For more information on getting objects by name, see "Representing Schemas" on page 2-11.

For use in displaying names or descriptions in a user interface, or for any purpose you want, you can associate any number of names and descriptions with an MdmObject by using the MdmDescription class. For information on using that class, see "Using Descriptions" on page 2-5.

Describing Unique Identifiers

Most metadata objects have a unique identifier (ID). The identifier has the form objectName or ownerName.objectName. For example, for the MdmDatabaseSchema that represents the schema for the user GLOBAL, the identifier returned by the getID method is GLOBAL. For an MdmPrimaryDimension named PRODUCT_AWJ, the getID method returns GLOBAL.PRODUCT_AWJ and for an MdmLevelHierarchy named PRODUCT_PRIMARY, the method returns GLOBAL.PRODUCT_AWJ.PRODUCT_PRIMARY. For some BaseMetadataObject objects, such as an MdmMetadataProvider, the getID method returns an empty string.

The ID of a metadata object is persistent. However, if the name or the owner of a metadata object changes, then the ID changes as well. For more information on getting objects by ID, see "Getting Metadata Objects by ID" on page 2-8.

For a legacy 10g metadata object, the first part of the identifier is a namespace. The namespace is followed by the namespace delimiter, which is two periods. An example of the identifier of a 10g dimension is AWXML_DIMENSION..GLOBAL.PRODUCT_AW.

Supporting Legacy Metadata Objects

In Oracle Database, Release 11*g*, Oracle Database, Release 11*g*Oracle OLAP supports legacy 10*g* OLAP Java API applications. Namespaces identify 10*g* metadata objects and enable them to exist in the same session as 11*g* objects.

Supporting Legacy Applications To support legacy applications that use OLAP metadata objects that were created in 10*g*, the oracle.olapi.data.source.DataProvider class has a metadata reader mode. By default, the metadata reader recognizes all Oracle OLAP 10*g* and 11*g* metadata objects. You can specify a metadata reader mode with a property of a java.util.Properties object or with a string in the proper XML format. For information on the modes and how to specify one, see the constructor methods of the DataProvider class in the *Oracle OLAP Java API Reference* documentation.

Describing Namespaces

In Oracle Database, Release 10*g*, an Oracle OLAP cube, dimension, or measure folder could have the same name as a relational table or view. In Release 11*g*, top-level OLAP metadata objects are stored in the Oracle Database data dictionary, so they cannot have the same name as another relational object. A namespace designation allows a legacy OLAP Java API 10*g* metadata object to exist in the same session as 11*g* metadata objects. Such legacy metadata objects were created by using classes in the oracle.olapi.AWXML package of the Oracle OLAP Analytic Workspace Java API or by using CWM PL/SQL packages. For 10*g* and 11*g* objects to exist in the same session, the metadata reader mode of the DataProvider must be set to ALL. The ALL mode is the default metadata reader mode. For more information on metadata reader mode settings, see the DataProvider class documentation in *Oracle OLAP Java API Reference*.

The metadata objects for a 10*g* cube, dimension, and measure folder are represented in 11*g* by the MdmCube, MdmPrimaryDimension, and MdmSchema classes. An instance of one of those classes can have a namespace associated with it, which is returned by the getNamespace method. For an 11*g* object, the namespace is null.

The 11*g* XML definition of a 10*g* object has a Namespace attribute. For information on exporting and importing XML definitions of metadata objects, see "Exporting and Importing Metadata as XML Templates" on page 2-8.

The namespace of a legacy metadata object identifies the metadata format and the type of object. It begins with either AWXML_ or CWM_ and then has the type of object, such as CUBE or DIMENSION. For example, a dimension created by using the Oracle OLAP Analytic Workspace Java API in Oracle Database 10*g*, Release 2 (10.2), would have the namespace AWXML_DIMENSION in 11*g*.

The valid namespaces are represented by static constant fields of the MdmMetadataProvider class. The getValidNamespaces method of that class returns a list of the valid namespaces, including the default namespace. You cannot create a new namespace.

You can use the constant fields to get a legacy metadata object from an MdmDatabaseSchema. For example, the following code gets the PRODUCT_AW dimension. In the code, mdmDBSchema is the MdmDatabaseSchema for the GLOBAL user.

```
MdmStandardDimension mdmProdAWDim =
  mdmDBSchema.findOrCreateStandardDimension("PRODUCT_AW",
    MdmMetadataProvider.AWXML_DIMENSION_NAMESPACE);
```

In the ALL metadata reader mode, you get an existing 10*g* metadata object but you cannot create a new one. If the legacy metadata object does not exist, the method returns an 11*g* object that has the specified name.

Using Descriptions

With an MdmDescription object, you can associate descriptive information with an MdmObject object. An MdmDescriptionType object represents the type of description of an MdmDescription. You can use MdmDescription objects to display names, descriptions, or other information for a metadata object in a user interface. MdmDescription objects are created, assigned, and handled entirely by your application.

Note: A descriptive name that you associate with an MdmObject through an MdmDescription is not the object name that is returned by the MdmObject.getName method. The object name is used by Oracle OLAP to identify the object internally. A descriptive name is used only by an application.

The OLAP Java API defines some types of descriptions. The `MdmDescriptionType` class has static methods that provide the following description types.

Name	Plural name	Description
Short name	Short plural name	Short description
Long name	Long plural name	Long description

You get one of these defined description types by calling a method of `MdmDescriptionType`. For example, the following code gets the description type object for a long name and a long description.

```
MdmDescriptionType mdmLongNameDescrType =
  MdmDescriptionType.getLongNameDescriptionType();
MdmDescriptionType mdmLongDescrDescrType =
  MdmDescriptionType.getLongDescriptionDescriptionType();
```

You can create a new type of description by using a constructor method of `MdmDescriptionType`. You can get the type of an `MdmDescriptionType` object with the `getDescriptiveType` method. Figure 2–2 shows the methods of `MdmDescriptionType`.

Some of the defined description types have an associated default description type. You change a default description type or assign a default description type for a new or existing `MdmDescriptionType` by using the `MdmDescriptionType(java.lang.String type, MdmDescriptionType defaultType)` constructor method. You can get the default type of an `MdmDescriptionType` object with the `getDescriptiveTypeDefault` method.

To associate an `MdmDescription` object with an `MdmObject`, use the `findOrCreateDescription` or a `setDescription` method of the `MdmObject`. The `findOrCreateDescription` method returns an `MdmDescription` object. To specify a value for the description, use the `setValue` method of `MdmDescription`.

Example 2–1 shows both ways of associating an `MdmDescription` with an `MdmObject`. In the example, `mdmProdDim` is an `MdmStandardDimension` object.

Example 2–1 Associating a Description with an MdmObject

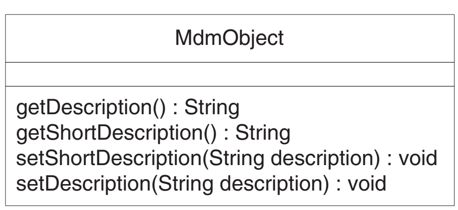
```
MdmDescription mdmShortNameDescr =
  mdmProdDim.findOrCreateDescription(
    MdmDescriptionType.getShortNameDescriptionType(), "AMERICAN");
mdmShortNameDescr.setValue("Product");

mdmProdDim.setDescription(
  MdmDescriptionType.getLongNameDescriptionType(), "Product Dimension");
```

Figure 2–2 shows the methods of `MdmObject` that use `MdmDescription` and `MdmDescriptionType` objects. It also shows the `MdmDescription` and `MdmDescriptionType` classes and their methods, and the associations between the classes.

Figure 2–2 MdmObject and MdmDescription Associations

Versions of the OLAP Java API before 11*g* did not have the `MdmDescription` and `MdmDescriptionType` classes. In those versions, the `MdmObject` class had only the following methods for getting and setting descriptions.

Figure 2–3 Methods for Getting and Setting Descriptions Before 11g



For backward compatibility, the OLAP Java API still supports these methods, but implements them internally using `MdmDescription` and `MdmDescriptionType` objects.

Using Classifications

A classification is a property of an `MdmObject`. You assign a classification to an object and then use the classification as you please. For example, you could add a classification with the value of "HIDDEN" to indicate that an application should not display the object in the user interface. You can assign a classification to an MdmObject by using the addObjectClassification method of the object. You can get a classification with the getObjectClassification method and remove one with the removeObjectClassification method.

Providing Metadata Objects

Access to Oracle OLAP Java API metadata objects is initially provided by an MdmMetadataProvider and by MdmSchema objects. The MdmMetadataProvider also has the ability to import or export an XML representation of a metadata object.

Describing Metadata Providers

Before you can get or create OLAP Java API metadata objects, you must first create an MdmMetadataProvider. For information on creating an MdmMetadataProvider, see "Creating an MdmMetadataProvider" on page 3-4.

With the getRootSchema method of the MdmMetadataProvider, you can get the root MdmSchema object, which is an instance of the MdmRootSchema class. The root schema is a container for MdmDatabaseSchema objects.

MdmDatabaseSchema objects are owners of top-level metadata objects such as AW, MdmCube, and MdmPrimaryDimension objects. The top-level objects are first class data objects and are represented in the Oracle Database data dictionary. You create top-level metadata objects by using methods of an MdmDatabaseSchema.

The top-level objects are the containers of objects such as MdmMeasure, MdmHierarchy, and MdmAttribute objects. You create the contained objects by using methods of the top-level objects.

For more information on MdmSchema objects, see "Representing Schemas" on page 2-11. For information on top-level metadata objects, see "Providing Access to Data Sources" on page 2-13.

You can also get an existing metadata object by calling the getMetadataObject or getMetadataObjects method of the MdmMetadataProvider and providing the ID of the metadata object.

Getting Metadata Objects by ID

Usually, you get or create metadata objects by calling methods on the owning object. For example, you can get or create an MdmCube by calling the findOrCreateCube method of an MdmDatabaseSchema object. However, you can also get an existing metadata object from an MdmMetadataProvider by specifying the ID of the object. The MdmMetadataProvider.getMetadataObject method takes a String that is the ID of an object and returns the object. The getMetadataObjects method takes a List of IDs and returns a List of objects.

You can store the ID of a metadata object from one session and then get the object by that ID in another session. Of course, getting an object by a stored ID assumes that the object still exists and that the ID of the object has not changed. For some metadata objects, you can change the name or the owner. If the name or owner of the object changes, then the ID of the object changes.

Exporting and Importing Metadata as XML Templates

The MdmMetadataProvider class has many methods for exporting and importing metadata objects to and from XML definitions of those objects. The XML definition is a template from which Oracle OLAP can create the metadata objects defined.

You can use XML templates to transport metadata objects between Oracle Database instances. You can exchange XML templates between Analytic Workspace Manager and an OLAP Java API application; that is, in Analytic Workspace Manager you can import a template that you created with an `MdmMetadataProvider` export XML method, and you can use an `importXML` method to import an XML template created by Analytic Workspace Manager.

When exporting XML, you can rename objects or specify bind variables for the values of XML attributes. You can also supply an implementation of the `XMLWriterCallback` interface to manage some aspects of the export process. When importing XML, you can specify an `MdmDatabaseSchema` to own the imported objects, bind values to replace the bind variables in the exported XML, and an implementation of the `XMLParserCallback` interface to manage some aspects of the import process.

Exporting XML Templates For exporting metadata objects to XML templates, `MdmMetadataProvider` has many signatures of the `exportFullXML` and `exportIncrementalXML` methods. The methods export a template to a `java.lang.String` or to a `java.io.Writer`.

You can use an XML template produced by these methods to import metadata objects through the `importXML` methods of `MdmMetadataProvider`. You can also use the XML template to import metadata objects in Analytic Workspace Manager.

An `exportFullXML` method exports the complete XML definitions for the specified objects or for the objects that you have created or modified since a specified `oracle.olapi.transaction.Transaction`. For an example of using the `exportFullXML` method, see Example 4–10, "Exporting to an XML Template".

An `exportIncrementalXML` method exports only the XML attributes that have changed for a metadata object since a specified `Transaction`. If you specify a `List` of objects, then the exported templates contain the XML attributes that have changed for the objects that are in the list, plus templates for any objects that are new since the specified `Transaction`. The exported incremental XML includes the type and name of the objects in the ownership and containment hierarchy of the changed object.

The `exportFullXML` and `exportIncrementalXML` methods take various combinations of the following parameters.

- A `List` of the objects to export or a `Transaction`.
- A `Writer` to which Oracle OLAP exports the XML. If you do not specify a `Writer`, then the method returns a `java.lang.String` that contains the XML.
- A `java.util.Map` that has metadata object references as keys and that has, as the objects for the keys, `String` values that contain new names for the referenced objects. With this `Map`, you can rename an object that you export. You can specify `null` for the parameter if you do not want to rename any objects.

  If you specify a `Map` for this `renameMap` parameter, then the Oracle OLAP XML generator renames a referenced object during the export. You can copy the definition of an existing object this way, by renaming an object during the export of an XML template and then importing the template.
- A `boolean` that specifies whether or not to include the name of the owning object in the exported XML.
- An optional `Map` that has metadata object references as keys and that has, as the objects for the keys, `String` values that function like SQL bind variables. For more information on the bind variables in this parameter, see "Describing Bind Variables in XML Templates" on page 2-11.

- An optional implementation of the `oracle.olapi.metadata.XMLWriterCallback` interface. With an `XMLWriterCallback`, you can specify whether or not to exclude an attribute or an owner name from the exported XML.

All metadata objects that share an ancestor are grouped together in the exported XML. For any object that is not a top-level object and whose top-level container is not in the `List` of the objects to export, the exported template contains an incremental definition to the object and a full definition below that. This supports the export of objects such as a calculated measure in a cube without having to export the entire cube template.

If an `MdmDatabaseSchema` is in the `List` of objects to export, then the exported template includes all objects within the database schema. If an `oracle.olapi.metadata.deployment.AW` object is in the `List`, then the exported template includes all of the objects that are contained by the `AW`. If the `MdmRootSchema` is in the list, it is ignored.

Importing XML Templates For importing metadata objects as XML templates, `MdmMetadataProvider` has several signatures of the `importXML` method.

An `importXML` method imports XML definitions of objects and either creates new objects or modifies existing objects. The `importXML` method take various combinations of the following parameters.

- A `java.io.Reader` for input of the XML or a `String` that contains the XML to import.
- An `MdmDatabaseSchema` to contain the new or modified metadata objects.
- A `boolean`, `modifyIfExists`, that indicates whether or not you want differences in the imported XML definition to modify an existing object of the same name.
- An optional `Map`, `bindValues`, that contains bind variables as keys and, as the objects for the keys, `String` values to replace the bind variables. For more information on the bind values in this parameter, see "Describing Bind Variables in XML Templates" on page 2-11.
- An optional implementation of the `oracle.olapi.metadata.XMLParserCallback` interface.

If the value of the `modifyIfExists` parameter is `true` and if the imported XML contains a full definition for an object that already exists and the object definition is different from the XML, then the method merges the new or changed elements of the object definition with the existing definition of the object. If the value of `modifyIfExists` is `false` and if the XML contains a full definition for an object that already exists and the object is different from the XML, then the `importXML` method throws an exception.

With the `bindValues` parameter, you can specify a `Map` that has key/object pairs that Oracle OLAP uses to replace bind variables when importing an XML template. A key in the `Map` is a bind variable to replace and the object paired to the key is the value with which to replace the bind variable. When you import a template, if you specify a `Map` that contains bind variables as keys, then Oracle OLAP replaces a bind variable in the imported XML with the value specified for the bind variable in the `bindValues` `Map`.

You can pass an implementation of the `XMLParserCallback` interface to an `importXML` method as the `parserCallback` parameter. With the `XMLParserCallback`, you can specify how Oracle OLAP handles an error that might occur when importing XML. The `XML11_2_ParserCallback` interface adds methods for renaming the imported object and for suppressing attributes of the imported object.

Describing Bind Variables in XML Templates

The `exportFullXML` and `exportIncrementalXML` methods have an optional `bindVariables` parameter. This parameter is a `Map` that has metadata objects as keys and `String` values as the objects for the keys. The `String` values function like SQL bind variables. During the export of the XML, the Oracle OLAP XML generator replaces the name of the referenced object with the bind variable.

If you provide a `Map` for the `bindVariables` parameter to an `exportFullXML` or `exportIncrementalXML` method, then the XML produced by the method begins with the following declaration.

```
<!DOCTYPE Metadata [
<!ENTITY % BIND_VALUES PUBLIC "OLAP BIND VALUES" "OLAP METADATA">
%BIND_VALUES;
 ]>
```

A value specified in the `bindVariables` map appears in the exported XML in the format `"&BV;"`, where `BV` is the bind variable.

The `bindValues` parameter of an `importXML` method specifies values that Oracle OLAP uses to replace the bind variables when importing an XML template. When you import a template, if you specify a `Map` that contains bind variables as keys, then Oracle OLAP replaces a bind variable in the imported XML with the `String` specified as the object for the bind variable key in the `Map`.

If you provide a `Map` for the `bindValues` parameter, then the `inXML` string that you provide to the method must include the `!DOCTYPE Metadata` declaration and the bind variables in the XML to import must be in the `"&BV;"` format.

Representing Schemas

Schemas are represented by the `MdmSchema` class and the subclasses of it. An `MdmSchema` is owner of, or a container for, `MdmCube`, `MdmDimension`, and other `MdmObject` objects, including other `MdmSchema` objects. In the 10*g* and earlier versions of the OLAP Java API, the `MdmSchema` class had more than one role. The API had one root `MdmSchema`, an `MdmSchema` for each measure folder, and custom `MdmSchema` objects that an application could create.

The 11*g* OLAP Java API introduced subclasses of `MdmSchema` to separate and define the different roles. In 11*g*, `MdmSchema` remains a concrete class for compatibility with the earlier versions and for use in 10*g* metadata reader modes.

In 11*g*, an `MdmSchema` is an instance of one of the following subclasses of `MdmSchema`:

- `MdmRootSchema`, which is a container for `MdmDatabaseSchema` objects and is supplied by the system.
- `MdmDatabaseSchema`, which represents the relational schema for a database user and which creates and owns `MdmCube`, `MdmDimension`, and other `MdmObject` objects. `MdmDatabaseSchema` objects are supplied by the system.
- `MdmOrganizationalSchema`, which you create to organize measures, cubes, dimensions, and other `MdmOrganizationalSchema` objects.

Some of the methods of `MdmSchema`, which are inherited by the subclasses, are nonfunctional in 11*g* and have been replaced by other methods. For example, for the `MdmRootSchema` class, the `getSubSchemas` method is replaced by getDatabaseSchemas, and for MdmDatabaseSchema, it is replaced by getOrganizationalSchemas.

This remainder of this topic describes the subclasses of MdmSchema.

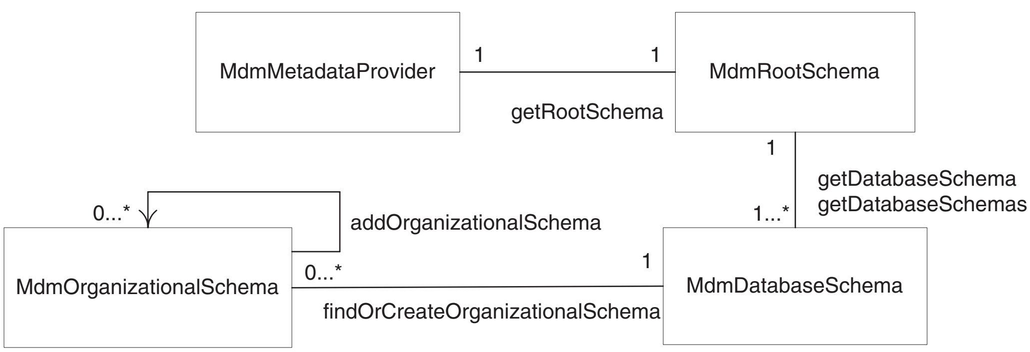
Representing the Root Schema

The root schema is a container for database schema objects. This top-level schema is represented by the MdmRootSchema class. You get the MdmRootSchema with the getRootSchema method of the MdmMetadataProvider. From the MdmRootSchema you can get all of the MdmDatabaseSchema objects or you can get an individual MdmDatabaseSchema by name.

The MdmRootSchema class also contains all of the MdmCube, MdmMeasure, and MdmPrimaryDimension objects that are provided by the MdmMetadataProvider, and has methods for getting those objects. However, the List of objects returned by those methods contains only the cubes, measures, or dimensions that the user has permission to see.

Figure 2–4 shows the associations between an MdmMetadataProvider and the subclasses of MdmSchema.

Figure 2–4 Associations Between MdmMetadataProvider and the MdmSchema Subclasses



Representing Database Schemas

Each Oracle Database user owns a relational schema. The schema for a database user is represented by the MdmDatabaseSchema class. The MdmRootSchema has one MdmDatabaseSchema object for each database user. An MdmDatabaseSchema has the same name as the database user. For example, the name of the MdmDatabaseSchema for the user GLOBAL is GLOBAL.

You can get one or all of the MdmDatabaseSchema objects with methods of the MdmRootSchema. However, access to the objects that are owned by an MdmDatabaseSchema is determined by the security privileges granted to the user of the session. For information on object and data security management and privileges, see *Oracle OLAP User's Guide*.

An MdmDatabaseSchema is the owner of top-level OLAP metadata objects and the objects created by them. You use an MdmDatabaseSchema to get existing metadata objects or to create new ones. The top-level objects are the following.

AW	MdmNamedBuildProcess	MdmPrimaryDimension
MdmCube	MdmOrganizationalSchema	MdmTable

Except for an MdmTable, you can create new top-level objects, or get existing ones, with the findOrCreate methods such as findOrCreateAW and findOrCreateStandardDimension. Creating objects is described in Chapter 3.

When you commit the Transaction in which you have created top-level OLAP metadata objects, those objects then exist in the Oracle data dictionary. They are available for use by ordinary SQL queries as well as for use by applications that use the Oracle OLAP Java API.

Because the metadata objects exist in the Oracle data dictionary, an Oracle Database DBA can restrict access to certain types of the metadata objects. In a client application, you can set such restrictions by using the JDBC API to send standard SQL GRANT and REVOKE commands through the JDBC connection for the user session.

You can get an MdmTable, or other top-level object, with the getTopLevelObject method. You can get all of instances of a particular type of top-level object with methods such as getAWs, getDimensions, or getOrganizationalSchemas, or you can use the getSchemaObjects to get all of the objects owned by the MdmDatabaseSchema. You can add or remove top-level objects with methods like addAW and removeSchemaObject.

Representing Organizational Schemas

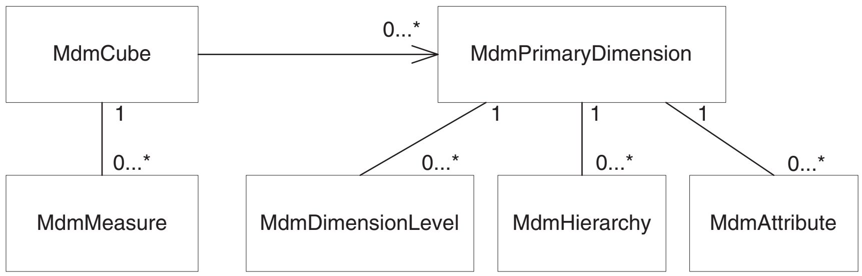
An OLAP measure folder organizes measures, cubes, and dimensions. A measure folder is represented by the MdmOrganizationalSchema class. Measure folders provide a way to differentiate among the similarly named measures. For example, a user may have access to several schemas with measures named Sales or Costs. You could separate measures that have the same name into different MdmOrganizationalSchema objects. An MdmOrganizationalSchema has methods for adding or removing cubes, dimensions, and measures. You can nest organizational schemas, so the class also has methods for adding and removing other MdmOrganizationalSchema objects.

Providing Access to Data Sources

Some of the classes in the mdm package that represent objects that contain or provide access to the data in the data store. Some of these classes represent OLAP dimensional data model objects, which include cubes, measures, dimensions, levels, hierarchies, and attributes. Other mdm classes represent relational objects such as tables, or columns in a view or table.

Figure 2–5 shows the associations between the classes that implement dimensional data model objects. An MdmCube can contain from zero to many MdmMeasure objects. An MdmMeasure is contained by one MdmCube object. An MdmCube can have from zero to many MdmPrimaryDimension objects, which are associated with it through MdmDimensionality objects. An MdmPrimaryDimension can contain from zero to many MdmDimensionLevel objects, MdmHierarchy objects, and MdmAttribute objects.

Figure 2–5 Associations of Dimensional Data Model Classes



The classes that represent these dimensional or relational data objects are subclasses of the `MdmSource` class. Subclasses of `MdmSource` have a `getSource` method, which returns a `Source` object. You use `Source` objects to define a query of the data. You then use `Cursor` objects to retrieve the data. For more information about working with `Source` and `Cursor` objects, see Chapter 5, "Understanding Source Objects" and Chapter 8, "Understanding Cursor Classes and Concepts".

You can also use SQL to query the views that Oracle OLAP automatically generates for the cubes, dimensions, and hierarchies. For information on querying these views, see "Getting Dimension and Hierarchy View and View Column Names" on page 2-27.

Representing Cubes and Measures

Cubes are the physical implementation of the dimensional model. They organize measures that have the same set of dimensions. Cubes and measures are dimensioned objects; the dimensions associated with a cube identify and categorize the data of the measures.

Representing Cubes

An OLAP cube is represented by the `MdmCube` class. An `MdmCube` is a container for `MdmMeasure` objects that are dimensioned by the same set of `MdmPrimaryDimension` objects. An application creates `MdmBaseMeasure` or `MdmDerivedMeasure` objects with the `findOrCreateBaseMeasure` and `findOrCreateDerivedMeasure` methods of an `MdmCube`. It associates each of the dimensions of the measures with the cube by using the `addDimension` method.

An `MdmCube` usually corresponds to a single fact table or view. To associate the table or view with the cube, you use `Query` and `CubeMap` objects. You get the `Query` for the table or view and then associate the `Query` with the `CubeMap` by using the `setQuery` method of the `CubeMap`.

The `CubeMap` contains `MeasureMap` and `CubeDimensionalityMap` objects that map the measures and dimensions of the cube to data sources. With the `MeasureMap`, you specify an `MdmBaseMeasure` and an `Expression` that identifies the column in the fact table or view that contains the base data for the measure.

To map the dimensions of the cube you get the `MdmDimensionality` objects of the cube. You create a `CubeDimensionalityMap` for each `MdmDimensionality`. You then specify an `Expression` for the `CubeDimensionalityMap` that identifies the foreign key column for the dimension in the fact table or view. If you want to specify a dimension column other than the column for the leaf level dimension members, then you must specify a join `Condition` with the `setJoinCondition` method of the `CubeDimensionalityMap`.

An MdmCube has an associated CubeOrganization. The CubeOrganization deploys the cube in an analytic workspace or as a relational database object. To deploy a cube to an analytic workspace, you call the findOrCreateAWCubeOrganization method of the MdmCube. You use the AWCubeOrganization returned by that method to specify characteristics of the cube, such as how Oracle OLAP builds the cube, how the cube stores measure data, and whether the database creates materialized views for the cube. For information on the AWCubeOrganization class, see *Oracle OLAP Java API Reference*.

If the AWCubeOrganization has a materialized view option of REWRITE_MV_OPTION, then Oracle OLAP creates a materialized view for the cube that can be used by the database query rewrite system. If the materialized view option is REWRITE_WITH_ATTRIBUTES_MV_OPTION, then Oracle OLAP includes in the rewrite materialized view the dimension attributes for which the isPopulateLineage method returns true. You set the materialized view options with the setMVOption method of the AWCubeOrganization.

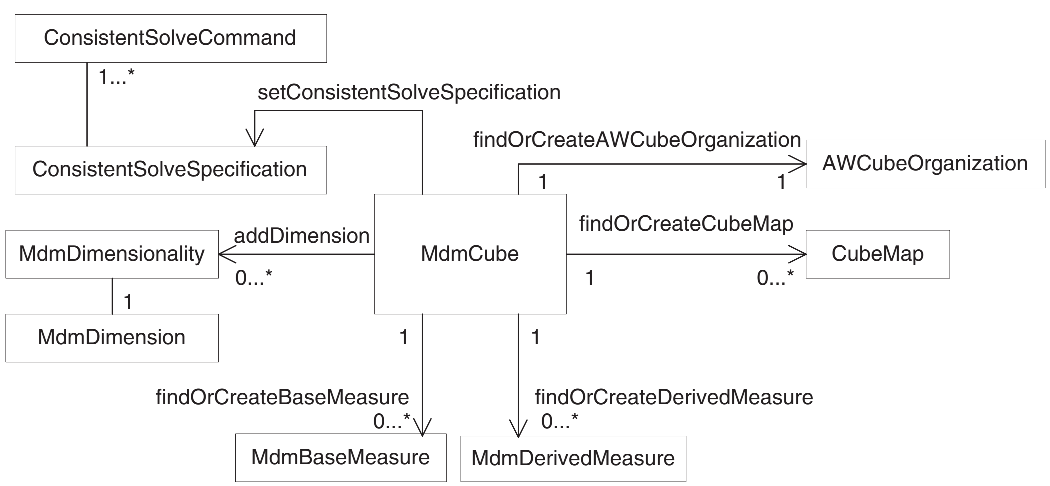
An MdmCube also has a ConsistentSolveSpecification object, which contains one or more ConsistentSolveCommand objects that specify how Oracle OLAP calculates (or *solves*) the values of the measures of the cube. For example, as the ConsistentSolveCommand, you could specify an AggregationCommand that represents the SUM or the MAX function. You specify the ConsistentSolveSpecification with the setConsistentSolveSpecification method of the cube.

A cube is *consistent* when the values of the measures match the specification, for example, when the values of the parents are equal to the SUM of the values of their children. A cube becomes consistent when the BuildProcess executes the ConsistentSolveCommand.

For examples of creating MdmCube and MdmMeasure objects and mapping them, and of the other operations described in this section, see Example 4–7, "Creating and Mapping an MdmCube" and Example 4–8, "Creating and Mapping Measures".

Figure 2–6 shows the associations between an MdmCube and the some of the classes mentioned in this section. The figure shows an MdmCube as deployed in an analytic workspace.

Figure 2–6 MdmCube and Associated Objects



Representing Measures

An MdmMeasure is an abstract class that represents a set of data that is organized by one or more MdmPrimaryDimension objects. The structure of the data is similar to that of a multidimensional array. Like the dimensions of an array, which provide the indexes for identifying a specific cell in the array, the MdmPrimaryDimension objects that organize an MdmMeasure provide the indexes for identifying a specific value of an element of the MdmMeasure.

For example, suppose you have an MdmMeasure that has data that records the number of product units sold to a customer during a time period and through a sales channel. The data of the measure is organized by dimensions for products, times, customers, and channels (with a channel representing the sales avenue, such as catalog or internet.). You can think of the data as occupying a four-dimensional array with the product, time, customer, and channel dimensions providing the organizational structure. The values of these four dimensions are indexes for identifying each particular cell in the array. Each cell contains a single data value for the number of units sold. You must specify a value for each dimension in order to identify a value in the array.

The values of an MdmMeasure are usually numeric, but a measure can have values of other data types. The concrete subclasses of MdmMeasure are MdmBaseMeasure and MdmDerivedMeasure.

An MdmBaseMeasure in an analytic workspace has associated physical storage structures. Typically an MdmCube gets the base data for an MdmBaseMeasure from a column in a fact table. Oracle OLAP then calculates the aggregate values of the measure and stores those values in an OLAP view for the cube.

An MdmDerivedMeasure has no associated physical storage. Oracle OLAP dynamically calculates the values for an MdmDerivedMeasure as needed.

The set of elements that are in an MdmMeasure is determined by the structure of the MdmPrimaryDimension objects of the MdmMeasure. That is, each element of an MdmMeasure is identified by a tuple, which is a unique combination of members from the MdmPrimaryDimension objects.

The MdmPrimaryDimension objects of an MdmMeasure are MdmStandardDimension or MdmTimeDimension objects. They usually have at least one hierarchical structure. Those MdmPrimaryDimension objects include all of the members of their component MdmHierarchy objects. Because of this structure, the values of the elements of an MdmMeasure are of one or more of the following:

- Values from the fact table column, view, or calculation on which the MdmMeasure is based. These values belong to MdmMeasure elements that are identified by a combination of values from the members at the leaf level of an MdmHierarchy.
- Aggregated values that Oracle OLAP has provided. These values belong to MdmMeasure elements that are identified by the value of at least one member from an aggregate level of an MdmHierarchy.
- Values specified by an Expression for a MdmDerivedMeasure or a custom dimension member.

As an example, imagine an MdmBaseMeasure called mdmUnitCost that is dimensioned by an MdmTimeDimension called mdmTimeDim and an MdmStandardDimension of products called mdmProdDim. Each of the mdmTimeDim and the mdmProdDim objects has all of the leaf members and aggregate members of the dimension it represents. A leaf member is one that has no children. An aggregate member has one or more children.

A unique combination of two members, one from mdmTimeDim and one from mdmProdDim, identifies each mdmUnitCost element, and every possible combination of dimension elements is used to specify the entire mdmUnitCost element set.

Some mdmUnitCost elements are identified by a combination of leaf members (for example, a particular product item and a particular month). Other mdmUnitCost elements are identified by a combination of aggregate members (for example, a particular product family and a particular quarter). Still other mdmUnitCost elements are identified by a mixture of leaf and aggregate members. The values of the mdmUnitCost elements that are identified only by leaf members come directly from the column in the database fact table (or fact table calculation). They represent the lowest level of data. However, for the elements that are identified by at least one aggregate member, Oracle OLAP provides the values. These higher-level values represent aggregated, or rolled-up, data.

Thus, the data represented by an MdmBaseMeasure is a mixture of fact table data from the data store and aggregated data that Oracle OLAP makes available for analytical manipulation. The data can include values that Oracle OLAP assigns as specified by an MdmModel.

Representing Dimensions, Levels, and Hierarchies

A dimension represents the general concept of a list of members that can organize a set of data. For example, if you have a set of figures that are the prices of product items during month time periods, then the unit price data is represented by an MdmMeasure that is dimensioned by dimensions for time and product values. The time dimension includes the month values and the product dimension includes item values. The month and item values act as indexes for identifying each particular value in the set of unit price data.

A dimension can contain levels and hierarchies. Levels can group dimension members into parent and child relationships, where members of lower levels are the children of parents that are in higher levels. Hierarchies define the relationships between the levels. Dimensions usually have associated attributes.

The base class for dimension, level, and hierarchy objects is the abstract class MdmDimension, which extends MdmSource. An MdmDimension has methods for getting and for removing the attributes associated with the object. It also has methods for getting and setting the cardinality and the custom order of the members of the object. The direct subclasses of MdmDimension are the abstract MdmPrimaryDimension and MdmSubDimension classes.

MdmPrimaryDimension and MdmHierarchyLevel objects can have associated MdmAttribute objects. For information on attributes, see "Representing Dimension Attributes" on page 2-21.

Representing Dimensions

Dimensions are represented by instances of the MdmPrimaryDimension class, which is an abstract subclass of MdmDimension. The concrete subclasses of the MdmPrimaryDimension class represent different types of data. The concrete subclasses of MdmPrimaryDimension are the following:

- MdmMeasureDimension, which has all of the MdmMeasure objects in the data store as the values of the dimension members. A data store has only one MdmMeasureDimension. You can obtain the MdmMeasureDimension by calling the getMeasureDimension method of the MdmRootSchema. You can get the measures of the data store by calling the getMeasures method of the MdmMeasureDimension.

- MdmStandardDimension, which has no special characteristics, and which typically represent dimensions of products, customers, distribution channels, and so on.
- MdmTimeDimension, which has time periods as the values of the members. Each time period has an end date and a time span. An MdmTimeDimension has methods for getting the attributes that record that information.

An MdmPrimaryDimension implements the following interfaces.

- Buildable, which is a marker interface for objects that you can specify in constructing a BuildItem.
- MdmMemberListMapOwner, which defines methods for finding or creating, or getting, a MemberListMap object.
- MdmViewColumnOwner, which is marker interface for objects that can have an associated MdmViewColumn.
- MetadataObject, which defines a method for getting a unique identifier.
- MdmQuery, which defines methods for getting the Query object associated with the implementing class and for getting information about the Query.

An MdmPrimaryDimension can have component MdmDimensionLevel objects that organize the dimension members into levels. It also can have MdmHierarchy objects, which organize the levels into the hierarchies. An MdmPrimaryDimension has all of the members of the component MdmHierarchy objects, while each of the MdmHierarchy objects has only the members in that hierarchy.

You can get all of the MdmPrimaryDimension objects that are contained by an MdmDatabaseSchema or an MdmOrganizationalSchema by calling the getDimensions method of the object. An MdmDatabaseSchema has methods for finding an MdmTimeDimension or an MdmStandardDimension by name or creating the object if it does not already exist.

MdmStandardDimension and MdmTimeDimension objects contain MdmAttribute objects. Some of the attributes are derived by Oracle OLAP, such as the parent attribute, and others you map to data in relational tables or to data that you specify by an Expression. For information on attributes, see "Representing Dimension Attributes" on page 2-21.

An MdmPrimaryDimension can organize the dimension members into one or more levels. Each level is represented by an MdmDimensionLevel object. An MdmStandardDimension or an MdmTimeDimension can contain MdmHierarchy objects that organize the levels into hierarchical relationships. In an MdmLevelHierarchy the dimension levels are represented by MdmHierarchyLevel objects. The concrete MdmDimensionLevel and MdmHierarchyLevel classes, and the abstract MdmHierarchy class, are the direct subclasses of the abstract MdmSubDimension class.

Representing Dimension Levels

An MdmDimensionLevel represents a set of dimension members that are at the same level. A dimension member can be in at most one dimension level. You get or create an MdmDimensionLevel with the findOrCreateDimensionLevel of an MdmPrimaryDimension. You can map an MdmDimensionLevel to a data source by using a MemberListMap.

An MdmPrimaryDimension has a method for getting a list of all of the MdmDimensionLevel objects that it contains. It also has a method for finding an MdmDimensionLevel by name or creating the object if it does not already exist.

Representing Hierarchies

MdmHierarchy is an abstract subclass of MdmSubDimension. The concrete subclasses of MdmHierarchy are MdmLevelHierarchy and MdmValueHierarchy.

An MdmHierarchy organizes the members of a dimension into a hierarchical structure. The parent-child hierarchical relationships of an MdmLevelHierarchy are based on the levels of the dimension. In an MdmValueHierarchy, the hierarchical relationships are based on dimension member values and not on levels. An MdmPrimaryDimension can have more than one of either or both kinds of hierarchies.

The parent of a hierarchy member is recorded in a parent MdmAttribute, which you can get by calling the getParentAttribute method of the MdmHierarchy. The ancestors of a hierarchy member are recorded in an ancestors MdmAttribute, which you can get by calling the getAncestorsAttribute method.

An MdmPrimaryDimension has a method for getting a list of all of the MdmHierarchy objects that it contains. It also has methods for finding an MdmLevelHierarchy or MdmValueHierarchy by name or creating the object if it does not already exist.

Representing a Level-based Hierarchy MdmLevelHierarchy is a subclass of MdmHierarchy. An MdmLevelHierarchy has a tree-like structure with a top, or highest, level, and a leaf, or lowest, level. Each member may have zero or one parent in the hierarchy. Cycles are not allowed, for example where member A is the parent of member B, member B is the parent of member C, and member C is the parent of member A.

Members that are not the child of any other member are the *top* members. Members with children are *aggregates* or *aggregate members* of the hierarchy. Members with no children are the *leaves* or *leaf members* of the hierarchy.

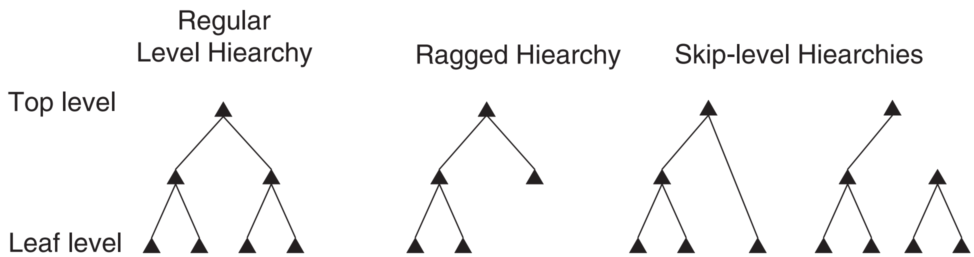
Each member is in a level. The levels are ordered, from top level to leaf level. The order is determined by the order in which you create the MdmDimensionLevel objects of the MdmPrimaryDimension. The first MdmDimensionLevel that you create is the top level and the last one you create is the leaf level. For example, for the CALENDAR_YEAR hierarchy of the TIME_AWJ dimension, the BuildAW11g example program creates four MdmDimensionLevel objects in the following order: TOTAL_TIME, YEAR, QUARTER, and MONTH. The top level is TOTAL_TIME and the leaf level is MONTH.

If a member of the hierarchy has a parent, then that parent must be in a higher level. Oracle OLAP expects that all leaf members in the hierarchy are in the leaf level. You can specify that Oracle OLAP allow the hierarchy to be *ragged*. In a ragged hierarchy, one or more leaf members are not in the leaf level. You can specify allowing the hierarchy to be ragged by calling the setIsRagged(true) method of the MdmLevelHierarchy.

Oracle OLAP also expects that if a member is in a level below the top level, then that member has a parent, and that the parent is in the level just above the level of the member. If a member is not at the top level and that member either does not have a parent or the parent is not in the next higher level, then the hierarchy is a skip-level hierarchy. You can specify allowing a skip-level hierarchy by calling the setIsSkipLevel(true) method of the MdmLevelHierarchy.

Figure 2–7 illustrates the relationships of members in a regular hierarchy, a ragged hierarchy, and two types of skip-level hierarchies.

Figure 2–7 Regular, Ragged, and Skip-level Hierarchies



The different levels of an `MdmLevelHierarchy` are represented by `MdmHierarchyLevel` objects. For an example of creating a level-based hierarchy, see "Creating and Mapping an MdmLevelHierarchy" on page 4-4.

The `MdmLevelHierarchy` has all of the members of the hierarchy, and each of the component `MdmHierarchyLevel` objects has only the members at the level that it represents. An `MdmLevelHierarchy` can also represent a nonhierarchical list of members, in which case the `MdmLevelHierarchy` has one `MdmHierarchyLevel`, and both objects have the same members. You get the levels of an `MdmLevelHierarchy` by calling the `getHierarchyLevels` method.

An `MdmLevelHierarchy` has a method for getting a list of all of the `MdmHierarchyLevel` objects that it contains. It also has a method for finding an `MdmHierarchyLevel` by name or creating the object if it does not already exist.

An `MdmPrimaryDimension` can contain more than one `MdmLevelHierarchy`. For example, an `MdmTimeDimension` dimension might have two `MdmLevelHierarchy` objects, one organized by calendar year time periods and the other organized by fiscal year time periods. The `MdmHierarchyLevel` objects of one hierarchy associate `MdmDimensionLevel` objects of calendar year time periods with the hierarchy. The `MdmHierarchyLevel` objects of the other hierarchy associate `MdmDimensionLevel` objects of fiscal year time periods with that hierarchy. Generally, level-based hierarchies share the lowest level, so the `MdmHierarchyLevel` for the lowest level of each of the hierarchies associates the same `MdmDimensionLevel` with each hierarchy. For example, the calendar year hierarchy and the fiscal year hierarchy share the same `MdmHierarchyLevel` of month time periods.

Representing a Value-based Hierarchy A value-based hierarchy is one in which levels are not meaningful in defining the hierarchical relationships. This type of hierarchy is represented by the `MdmValueHierarchy` class, which is a subclass of `MdmHierarchy`. An example of a value hierarchy is the employee reporting structure of a company, which can be represented with parent-child relationships but without levels. For an example of creating a value-based hierarchy, see "Creating and Mapping an MdmValueHierarchy" on page 4-5.

The OLAP view for the value hierarchy has a column that contains all employees, including those who are managers. It has another column that contains the parent members. Another column identifies the depth of the member in the hierarchy, where the member that has no manager is at depth 0 (zero), the employees who report to that manager are at level 1, and so on.

Representing Hierarchy Levels

`MdmHierarchyLevel` is a subclass of `MdmSubDimension`. An `MdmHierarchyLevel` associates an `MdmDimensionLevel` with an `MdmLevelHierarchy`.

The order of the levels in the hierarchy is specified by the order in which you create the MdmHierarchyLevel objects for the MdmLevelHierarchy. The first MdmHierarchyLevel that you create is the highest level and the last one that you create is the lowest level. For an example of creating a hierarchy, see "Creating and Mapping an MdmLevelHierarchy" on page 4-4.

Representing Dimension Attributes

An OLAP dimension attribute is represented by an MdmAttribute object. An MdmAttribute has values that are related to members of an MdmPrimaryDimension. The MdmAttribute class is a subclass of MdmDimensionedObject because, like an MdmMeasure, the values of an MdmAttribute have meaning in relation to the members of the dimension.

The relation can be one-to-one, many-to-one, or one-to-many. For example, the Product dimension has a short description attribute, a package attribute, and an ancestors attribute. The short description attribute has a separate value for each dimension member. The package attribute has a set of values, each of which applies to more than one dimension member. The ancestors attribute has multiple values that apply to a single dimension member. If an MdmAttribute does not apply to a member of an MdmDimension, then the MdmAttribute value for that member is null.

Table 2–3 shows the first few members of the Product dimension and their related short description and package attribute values. Only some of the members of the Item level of the dimension have a package attribute. For other items, and for higher levels, the package attribute value is null, which appears as NA in the table.

Table 2–3 Dimension Members and Related Attribute Values

Dimension Member	Related Short Description	Related Package
TOTAL_PRODUCT::TOTAL	Total Product	NA
CLASS::HRD	Hardware	NA
FAMILY::DISK	CD/DVD	NA
ITEM::EXT CD ROM	External 48X CD-ROM	NA
ITEM::EXT DVD	External - DVD-RW - 8X	Executive
ITEM::INT 8X DVD	Internal - DVD-RW - 8X	NA
ITEM::INT CD ROM	Internal 48X CD-ROM	Laptop Value Pack
ITEM::INT CD USB	Internal 48X CD-ROM USB	NA
ITEM::INT RW DVD	Internal - DVD-RW - 6X	Multimedia
...	...	...

To get values from an MdmAttribute, you must join the Source for the MdmAttribute and the Source that specifies one or more members of the MdmDimension. For examples of joining the Source objects for an MdmAttribute and an MdmDimension, see Example 4–5 and examples from Chapter 6 such as Example 6–10.

Describing the MdmAttribute Class

The abstract MdmAttribute class has a subclass, which is the abstract class MdmSingleValuedAttribute. That class has two concrete subclasses: MdmBaseAttribute and MdmDerivedAttribute.

Specifying a Data Type When you create an MdmAttribute, you can specify the SQL data type. If you do not specify it, then the MdmAttribute has the data type of the source data that you map to it. For example, the SQL data type of the short description attribute is VARCHAR2 and the data type of the end date attribute is DATE.

Describing Types of Attributes An MdmAttribute is contained by the MdmPrimaryDimension that creates it. Some attributes, such as the parent attribute and the level attribute, are derived by Oracle OLAP from the structure of the dimension. Others are common attributes for which an MdmPrimaryDimension has accessor methods, such as the long and short description attributes, or the end date and time span attributes that an MdmTimeDimension requires. After you create one of those attributes, you associate it with the dimension through a method such as the setShortValueDescriptionAttribute method of an MdmPrimaryDimension or the setTimeSpanAttribute method of an MdmTimeDimension. You can also create attributes for your own purposes, such as the package attribute.

Associating an Attribute with an MdmSubDimension After you create an attribute, you associate it with an MdmSubDimension. You can associate it with just one MdmSubDimension by using the addAttribute method of the MdmSubDimension. You can also associate it with all of the MdmDimensionLevel objects of an MdmPrimaryDimension by using the setIsVisibleForAll method of the MdmAttribute. If you specify true with the setIsVisibleForAll method, then the attribute applies to all of the MdmDimensionLevel objects that are currently contained by the MdmPrimaryDimension and to any MdmDimensionLevel objects that you subsequently create or add to the dimension.

Getting MdmAttribute Objects The getAttributes method of an MdmPrimaryDimension returns all of the MdmAttribute objects that were created by a client application. The getAttributes method of an MdmSubDimension returns only those attributes that the application added to it with it the addAttribute method. Other methods of an MdmPrimaryDimension return specific attributes that Oracle OLAP generates, such as the getHierarchyAttribute, the getLevelDepthAttribute, or the getParentAttribute method.

Specifying a Target Dimension A target dimension for an attribute is similar to defining a foreign key constraint between columns in a table. All of the values of the attribute must also be keys of the target dimension.

You can specify a target dimension for an attribute by using the setTargetDimension method of the MdmAttribute. The relational table that is the Query for the target dimension must have a column that contains all of the values that are in the column of the dimension table to which you map the attribute.

Describing the MdmBaseAttribute Class

An MdmBaseAttribute has values that are stored in the OLAP views for the dimension that contains it and the hierarchy to which it applies. For information on OLAP views, see "Using OLAP Views" on page 2-26.

You create an MdmBaseAttribute with the findOrCreateBaseAttribute method of an MdmPrimaryDimension. You map the MdmBaseAttribute to a column in a relational table or view. When you build the MdmPrimaryDimension that created the attribute, Oracle OLAP stores the values of the MdmBaseAttribute in an OLAP view. You can get the column for the MdmBaseAttribute in the OLAP view by using the getETAttributeColumn method. That method returns an MdmViewColumn object.

Examples of MdmBaseAttribute objects are the name attribute created and mapped in Example 4–5 and the long description attribute created in Example 4–6. The mapping for that long description attribute is in Example 4–3.

For regular OLAP queries, using Source objects, you only need to map an MdmBaseAttribute to MdmDimensionLevel objects by using MemberListMap objects. For SQL queries against OLAP views, you should map the attributes to MdmHierarchyLevel objects by using HierarchyLevelMap objects.

Grouping Attributes With the setAttributeGroupName method of an MdmBaseAttribute, you can specify a name for an attribute group. You can specify the same group name for other attributes. For example, you could create a long description attribute for each dimension level and give each attribute the group name of LONG_DESCRIPTION. You could use the group name to identify similar kinds of attributes. You get the group name with the getAttributeGroupName method.

Creating an Index You can improve the performance of attribute-based queries by creating an index for the attribute. Creating an index adds maintenance time and increases the size of the analytic workspace, which may increase the build time for extremely large dimensions. You create an index for an attribute by specifying true wit the setCreateAttributeIndex method of the AWAttributeOrganization for the MdmBaseAttribute.

Specifying a Language for an Attribute When you create an AttributeMap for an MdmBaseAttribute, you can specify a language for the attribute. For example, to specify French as the language for the long description attribute for the MdmDimensionLevel named CHANNEL, you would create an AttributeMap by calling the MemberListMap.findOrCreateAttributeMap method and passing in the long description MdmBaseAttribute and FRENCH as the String that specifies the language. You would then specify GLOBAL.CHANNEL_DIM.CHANNEL_DSC_FRENCH as the Expression for the AttributeMap. By using the setLanguage method of an AttributeMap, you can specify a language for an AttributeMap after you have created it.

Specifying Multilingual Attributes The MdmBaseAttribute.setMultiLingual method allows you to map more than one language column to the same attribute. To do so, you specify true with the setMultiLingual method of the attribute. You then create a separate AttributeMap for each language but you use the same MdmBaseAttribute.

The language in use for the database determines which language appears in the OLAP view for the dimension. Only one language is in use at a time in a session, but if the language in use changes, then the language in the attribute column in the OLAP view also changes. For more information on specifying languages for database sessions, see *Oracle Database Globalization Support Guide*.

For materialized views, you should create a separate attribute for each language, so that there is a long description attribute for English, one for French, and so on. That behavior is more typical in SQL, which does not expect multivalued columns.

Populating OLAP Views with Hierarchical Attribute Values For SQL queries, you should populate the lineage of the attributes in the view by specifying true with the MdmBaseAttribute.setPopulateLineage method. Populating the lineage means that in the column for an attribute in an OLAP view, Oracle OLAP populates the rows for lower levels in a dimension hierarchy with the attribute values that are mapped at a higher level. Populating the lineage for the attributes is also useful if you are creating materialized views for an analytic workspace cube.

If you specify `setPopulateLineage(false)`, which is the default for the setting, then the attribute values appear only in the rows for the dimension members at the level to which the attribute is mapped. For dimension members at other levels, the attribute value is `null`. If you specify `setPopulateLineage(true)`, then the attribute values appear in the rows for the members of the mapped level and for the dimension members of all levels that are descendants of the mapped level.

Populating the hierarchy lineage in an OLAP view makes the contents of the view more like the contents of a relational table in a star schema. For example, you could create a separate long description attribute on the dimension for each `MdmDimensionLevel` of the dimension. You would specify populating the lineage of those attributes by calling the `setPopulateLineage(true)` method of each attribute. You would then make the attribute visible for a dimension level by adding the attribute to the `MdmDimensionLevel` with the `addAttribute` method.

The OLAP view for a hierarchy of the dimension would then have a column for each of the long description attributes. Those columns would contain the long description attribute values for the members of the mapped dimension level and for the dimension members of all levels that are descendants of the mapped level.

For example, the `BuildAW11g` example program has a line of code that specifies populating the lineage for the `MdmBaseAttribute` objects that it creates for each individual `MdmDimensionLevel`. The following line appears in the `createAttribute` method of the program.

```
mdmAttr.setPopulateLineage(true);
```

Example 2–2 shows the results of the following SQL query when that line of code is commented out. Example 2–3 shows the results of the SQL query when the line is included in the program. Both examples show the values that are in the selected columns of the OLAP view for the `PRODUCT_PRIMARY` hierarchy. The view name is `PRODUCT_AWJ_PRODUCT_PRIMA_VIEW`. The examples show only a few of the lines returned by the SQL query.

```
SELECT TOTAL_PRODUCT_SHORT_DESC || '*' || CLASS_SHORT_DESC || '*' ||
        FAMILY_SHORT_DESC || '*' || ITEM_SHORT_DESC
  FROM PRODUCT_AWJ_PRODUCT_PRIMA_VIEW
  ORDER BY TOTAL_PRODUCT nulls first, CLASS nulls first,
           FAMILY nulls first, ITEM nulls first;
```

In Example 2–2, the attribute rows of the OLAP view have only the attribute values for the hierarchy level to which the dimension member belongs.

Example 2–2 Values in OLAP View Columns After setPopulateLineage(false)

```
TOTAL_PRODUCT_SHORT_DESC||'*'||CLASS_SHORT_DESC||'*'||FAMILY_SHORT_DESC||'*'||IT
--------------------------------------------------------------------------------
Total Product***
*Hardware**
**CD/DVD*
***External 48X CD-ROM
***External - DVD-RW - 8X
***Internal - DVD-RW - 8X
...
**Desktop PCs*
***Sentinel Financial
***Sentinel Multimedia
***Sentinel Standard
**Portable PCs*
***Envoy Ambassador
```

```
***Envoy Executive
***Envoy Standard
...
```

In Example 2–3, the attribute rows of the OLAP view are populated with the attribute values for the ancestors of a dimension member. For example, the first row contains only the value Total Product because `TOTAL_PRODUCT` is the highest level in the hierarchy. The row that contains the value `Envoy Standard` also has the values for the `TOTAL_PRODUCT`, `CLASS`, and `FAMILY` levels.

Example 2–3 Values in OLAP View Columns After setPopulateLineage(true)

```
TOTAL_PRODUCT_SHORT_DESC||'*'||CLASS_SHORT_DESC||'*'||FAMILY_SHORT_DESC||'*'||IT
--------------------------------------------------------------------------------
Total Product***
Total Product*Hardware**
Total Product*Hardware*CD/DVD*
Total Product*Hardware*CD/DVD*External 48X CD-ROM
Total Product*Hardware*CD/DVD*External - DVD-RW - 8X
Total Product*Hardware*CD/DVD*Internal - DVD-RW - 8X
...
Total Product*Hardware*Desktop PCs*
Total Product*Hardware*Desktop PCs*Sentinel Financial
Total Product*Hardware*Desktop PCs*Sentinel Multimedia
Total Product*Hardware*Desktop PCs*Sentinel Standard
Total Product*Hardware*Portable PCs*
Total Product*Hardware*Portable PCs*Envoy Ambassador
Total Product*Hardware*Portable PCs*Envoy Executive
Total Product*Hardware*Portable PCs*Envoy Standard
...
```

Preparing Attributes for Materialized Views To generate materialized views for the OLAP metadata objects, for each `MdmDimensionLevel` you must create an `MdmBaseAttribute`, map it to a unique key for the `MdmDimensionLevel`, and add it to the `MdmDimensionLevel`. An `MdmDimensionLevel` has methods for adding, getting, and removing unique key attributes. The `EnableMVs.java` example program creates unique key attributes and adds them to the `MdmDimensionLevel` objects of the dimensions. For information about using materialized views, see *Oracle OLAP User's Guide*.

When Oracle OLAP creates a materialized view for a cube, it creates columns for the attributes of the dimensions of the cube. For the name of a column, it uses the name of the attribute column from the OLAP view of the dimension. To insure that the column name is unique, Oracle OLAP adds a default prefix to the name. You can specify the prefix by using the `setETAttrPrefix` method of the `MdmDimensionality` object for a dimension of the cube.

Describing the MdmDerivedAttribute Class

An `MdmDerivedAttribute` has values that Oracle OLAP calculates on the fly as you need them. You create an `MdmDerivedAttribute` with the `findOrCreateDerivedAttribute` method of an `MdmPrimaryDimension`. The values of an `MdmDerivedAttribute` come from a relation between dimension members or from an `Expression` that you specify. Oracle OLAP generates several `MdmDerivedAttribute` objects, such as the attributes returned by the `getParentAttribute` and the `getAncestorsAttribute` methods of an `MdmPrimaryDimension`.

Using OLAP Views

For each instance of an MdmCube, MdmPrimaryDimension, and MdmHierarchy in an analytic workspace, Oracle OLAP automatically creates an associated relational view. Oracle OLAP uses these views internally to provide access to the aggregate and calculated data that is generated by the analytic workspace. An OLAP Java API query transparently uses the views. In the OLAP Java API, these views are called ET (embedded totals) views.

A SQL application can directly query these views, using them as it would the fact tables and dimension tables of a star or snowflake schema. The *Oracle OLAP User's Guide* documentation refers to these views as OLAP views and describes them in detail. For those detailed descriptions, see *Oracle OLAP User's Guide*.

A client OLAP Java API application can get the names of the OLAP views and get the names of columns in the views. The application could display the names to the end user of the application, and the end user could then use the names in a SQL SELECT statement to query the OLAP objects.

Getting Cube View and View Column Names

To get the name of a cube view, call the MdmCube.getViewName() method. For example, the following code gets the name of the view for the MdmCube that is named UNITS_CUBE_AWJ. In the code, the mdmDBSchema object is the MdmDatabaseSchema for the GLOBAL user.

```
MdmCube mdmUnitsCube =
 (MdmCube)mdmDBSchema.getTopLevelObject("UNITS_CUBE_AWJ");
String cubeViewName = mdmUnitsCube.getViewName();
println("The name of the view for the " +
         mdmUnitsCube.getName() + " cube is " + cubeViewName + ".");
```

The output of the code is the following.

```
The name of the view for the UNITS_CUBE_AWJ cube is UNITS_CUBE_AWJ_VIEW.
```

You can change the name of the OLAP view by using the MdmCube.setViewName method. To make the name change permanent, you must commit the Transaction.

The OLAP view for an MdmCube has a column for each measure of the cube, including each derived measure. In *Oracle OLAP User's Guide*, a derived measure is known as a calculated measure. A cube view also has a column for each dimension of the cube. For example, for the MdmCube named UNITS_CUBE_AWJ, the view is named UNITS_CUBE_AWJ_VIEW. The following code gets the names of the view columns.

```
MdmCube mdmUnitsCube = mdmDBSchema.findOrCreateCube("UNITS_CUBE_AWJ");
List<MdmQueryColumn> mdmQCols = mdmUnitsCube.getQueryColumns();
for (MdmQueryColumn mdmQCol : mdmQCols )
{
  MdmViewColumn mdmViewCol = (MdmViewColumn) mdmQCol;
  println(mdmViewCol.getViewColumnName());
}
```

The code displays the following output.

```
TIME_AWJ
PRODUCT_AWJ
CUSTOMER_AWJ
CHANNEL_AWJ
UNITS
SALES
COST
```

The UNITS, SALES, and COST columns are for the measures of the cube, and the other four columns are for the dimensions of the cube.

Getting Dimension and Hierarchy View and View Column Names

To get the name of the OLAP view for a dimension or a hierarchy, call the getETViewName() method of the MdmPrimaryDimension or MdmHierarchy. You can get the name of a column in the view by calling the appropriate method of the metadata object. For example, the following code gets the name of the key column for the CHANNEL_AWJ dimension and the parent column for the CHANNEL_PRIMARY hierarchy.

```
println(mdmChanDim.getETKeyColumn().getViewColumnName());
MdmViewColumn mdmParentCol =(MdmViewColumn) mdmChanHier.getETParentColumn();
println(mdmParentCol.getViewColumnName());
```

The code displays the following output.

```
DIM_KEY
PARENT
```

You can change the name of the OLAP view by using the setETViewName method of the MdmPrimaryDimension or MdmHierarchy.

The OLAP view for an MdmPrimaryDimension has a column for the dimension keys, a column for each dimension level, and a column for each attribute associated with the dimension. For example, for the MdmStandardDimension named CHANNEL_AWJ, the view is named CHANNEL_AWJ_VIEW. The SQL command DESCRIBE CHANNEL_AWJ_VIEW displays the names of the following columns.

```
DIM_KEY
LEVEL_NAME
MEMBER_TYPE
DIM_ORDER
LONG_DESCRIPTION
SHORT_DESCRIPTION
TOTAL_CHANNEL_LONG_DESC
TOTAL_CHANNEL_SHORT_DESC
CHANNEL_LONG_DESC
CHANNEL_SHORT_DESC
```

The OLAP view for an MdmHierarchy has a column for the dimension keys and a column for the parent of a hierarchy member. If it is an MdmLevelHierarchy, then it also has a column for each hierarchy level and a column for the depth of a level. If the hierarchy has one or more added attributes, then the view has a column for each attribute. For example, for the MdmLevelHierarchy named CHANNEL_PRIMARY, the view is named CHANNEL_AWJ_CHANNEL_PRIMA_VIEW. The SQL command DESCRIBE CHANNEL_AWJ_CHANNEL_PRIMA_VIEW displays the names of the following columns.

```
DIM_KEY
LEVEL_NAME
MEMBER_TYPE
DIM_ORDER
HIER_ORDER
LONG_DESCRIPTION
SHORT_DESCRIPTION
TOTAL_CHANNEL_LONG_DESC
TOTAL_CHANNEL_SHORT_DESC
CHANNEL_LONG_DESC
```

```
CHANNEL_SHORT_DESC
PARENT
DEPTH
TOTAL_CHANNEL
CHANNEL
```

Using OLAP View Columns

The "Creating Basic Queries" topic of *Oracle OLAP User's Guide* provides several examples of how to create SQL queries using the OLAP views. An OLAP Java API query that uses `Source` objects automatically uses these views.

You can also provide direct access to the OLAP views to the users of your OLAP Java API application. You could allow users to specify a SQL `SELECT` statement that uses the views and then send that SQL query to the database.

Example 2–4 reproduces Example 4-2 of *Oracle OLAP User's Guide* except that it uses the cubes and dimensions of the `GLOBAL_AWJ` analytic workspace. The example selects the Sales measure from `UNITS_CUBE_AWJ_VIEW`, and joins the keys from the cube view to the hierarchy views to select the data.

In the example, `mdmDBSchema` is the `MdmDatabaseSchema` for the GLOBAL user. The example is an excerpt from the `BasicCubeViewQuery.java` example program.

Example 2–4 Basic Cube View Query

```
// In a method...
  // Get the cube.
  MdmCube mdmUnitsCube =
     MdmCube)mdmDBSchema.getTopLevelObject("UNITS_CUBE_AWJ");
  // Get the OLAP view for the cube.
  String cubeViewName = mdmUnitsCube.getViewName();
  // Display the name of the OLAP view for the cube.
  println("The name of the OLAP view for the " + mdmUnitsCube.getName()
          + " cube is:\n  " + cubeViewName);

  // Get the dimensions and the hierarchies of the dimensions.
  MdmPrimaryDimension mdmTimeDim =
   (MdmPrimaryDimension)mdmDBSchema.getTopLevelObject("TIME_AWJ");
  MdmLevelHierarchy mdmCalHier =
    mdmTimeDim.findOrCreateLevelHierarchy("CALENDAR_YEAR");

  // Display the name of the OLAP view name for the hierarchy and
  // display the names of the hierarchy levels.
  displayViewAndLevelNames(mdmCalHier);

  MdmPrimaryDimension mdmProdDim =
   (MdmPrimaryDimension)mdmDBSchema.getTopLevelObject("PRODUCT_AWJ");
  MdmLevelHierarchy mdmProdHier =
     mdmProdDim.findOrCreateLevelHierarchy("PRODUCT_PRIMARY");
  displayViewAndLevelNames(mdmProdHier);

  MdmPrimaryDimension mdmCustDim =
   (MdmPrimaryDimension)mdmDBSchema.getTopLevelObject("CUSTOMER_AWJ");
  MdmLevelHierarchy mdmShipHier =
    mdmCustDim.findOrCreateLevelHierarchy("SHIPMENTS");
  displayViewAndLevelNames(mdmShipHier);

  MdmPrimaryDimension mdmChanDim =
   (MdmPrimaryDimension)mdmDBSchema.getTopLevelObject("CHANNEL_AWJ");
  MdmLevelHierarchy mdmChanHier =
```

```
    mdmChanDim.findOrCreateLevelHierarchy("CHANNEL_PRIMARY");
  displayViewAndLevelNames(mdmChanHier);

  // Create a SQL SELECT statement using the names of the views and the
  // levels.
  // UNITS_CUBE_AWJ_VIEW has a column named SALES for the sales measure.
  // TIME_AWJ_CALENDAR_YEAR_VIEW has a column named LONG_DESCRIPTION
  // for the long description attribute.
  // The hierarchy views have columns that have the same names as the levels.
  String sql = "SELECT t.long_description time,\n" +
    "     ROUND(f.sales) sales\n" +
    "  FROM TIME_AWJ_CALENDAR_YEAR_VIEW t,\n" +
    "     PRODUCT_AWJ_PRODUCT_PRIMA_VIEW p,\n" +
    "     CUSTOMER_AWJ_SHIPMENTS_VIEW cu,\n" +
    "     CHANNEL_AWJ_CHANNEL_PRIMA_VIEW ch,\n" +
    "     UNITS_CUBE_AWJ_VIEW f\n" +
    "  WHERE t.level_name = 'YEAR'\n" +
    "     AND p.level_name = 'TOTAL_PRODUCT'\n" +
    "     AND cu.level_name = 'TOTAL_CUSTOMER'\n" +
    "     AND ch.level_name = 'TOTAL_CHANNEL'\n" +
    "     AND t.dim_key = f.time_awj\n" +
    "     AND p.dim_key = f.product_awj\n" +
    "     AND cu.dim_key = f.customer_awj\n" +
    "     AND ch.dim_key = f.channel_awj\n" +
    "  ORDER BY t.end_date";

  // Display the SQL SELECT statement.
  println("\nThe SQL SELECT statement is:\n" + sql);

  // Display the results of the SQL query.
  String title = "\nThe results of the SQL query are:\n";
  executeSQL(sql, title);
// ...
} // End of method.

private void displayViewAndLevelNames(MdmLevelHierarchy mdmLevelHier)
{
  // Get the OLAP view name for the hierarchy.
  String levelHierViewName = mdmLevelHier.getETViewName();
  // Display the name of the OLAP view for the hierarchy.
  println("\nThe OLAP view for the " + mdmLevelHier.getName() +
          " hierarchy is:\n  " + levelHierViewName);

  // Display the names of the levels of the hierarchy.
  displayLevelNames(mdmLevelHier);
}

private void displayLevelNames(MdmLevelHierarchy mdmLevelHier)
{
  List<MdmHierarchyLevel> mdmHierLevelList =
    mdmLevelHier.getHierarchyLevels();
 println("The names of the levels of the "
         + mdmLevelHier.getName() + " hierarchy are:");
 for (MdmHierarchyLevel mdmHierLevel :  mdmHierLevelList)
 {
   println("  " + mdmHierLevel.getName());
 }
}
```

```
protected void executeSQL(String sql, String heading)
{
  try
  {
    Statement statement = dp.getConnection().createStatement();
    println(heading);
    ResultSet rs = statement.executeQuery(sql);
    SQLResultSetPrinter.printResultSet(getCursorPrintWriter(), rs);
    rs.close();
    statement.close();
  }
  catch (SQLException e)
  {
    println("Could not execute SQL statement. " + e);
  }
}
```

The output of Example 2–4 is the following.

```
The name of the OLAP view for the UNITS_CUBE_AWJ cube is:
  UNITS_CUBE_AWJ_VIEW

The OLAP view for the CALENDAR_YEAR hierarchy is:
  TIME_AWJ_CALENDAR_YEAR_VIEW
The names of the levels of the CALENDAR_YEAR hierarchy are:
  TOTAL_TIME
  YEAR
  QUARTER
  MONTH

The OLAP view for the PRODUCT_PRIMARY hierarchy is:
  PRODUCT_AWJ_PRODUCT_PRIMA_VIEW
The names of the levels of the PRODUCT_PRIMARY hierarchy are:
  TOTAL_PRODUCT
  CLASS
  FAMILY
  ITEM

The OLAP view for the SHIPMENTS hierarchy is:
  CUSTOMER_AWJ_SHIPMENTS_VIEW
The names of the levels of the SHIPMENTS hierarchy are:
  TOTAL_CUSTOMER
  REGION
  WAREHOUSE
  SHIP_TO

The OLAP view for the CHANNEL_PRIMARY hierarchy is:
  CHANNEL_AWJ_CHANNEL_PRIMA_VIEW
The names of the levels of the CHANNEL_PRIMARY hierarchy are:
  TOTAL_CHANNEL
  CHANNEL

The SQL SELECT statement is:
SELECT t.long_description time,
     ROUND(f.sales) sales
  FROM TIME_AWJ_CALENDAR_YEAR_VIEW t,
     PRODUCT_AWJ_PRODUCT_PRIMA_VIEW p,
     CUSTOMER_AWJ_SHIPMENTS_VIEW cu,
     CHANNEL_AWJ_CHANNEL_PRIMA_VIEW ch,
     UNITS_CUBE_AWJ_VIEW f
```

```
  WHERE t.level_name = 'YEAR'
     AND p.level_name = 'TOTAL_PRODUCT'
     AND cu.level_name = 'TOTAL_CUSTOMER'
     AND ch.level_name = 'TOTAL_CHANNEL'
     AND t.dim_key = f.time_awj
     AND p.dim_key = f.product_awj
     AND cu.dim_key = f.customer_awj
     AND ch.dim_key = f.channel_awj
  ORDER BY t.end_date
```

The results of the SQL query are:

```
TIME       SALES
---------- ----------------------
1998       100870877
1999       134109248
2000       124173522
2001       116931722
2002       92515295
2003       130276514
2004       144290686
2005       136986572
2006       140138317
2007       <null>
```

Using Source Objects

Example 2–4 demonstrates how to create a SQL statement using the OLAP views. You can produce the same results by using OLAP Java API `Source` objects, as shown in Example 2–5. The code in Example 2–5 uses the `MdmLevelHierarchy` objects from Example 2–4.

Example 2–5 Basic Cube Query Using Source Objects

```
// Get the Sales measure and the Source for it.
MdmBaseMeasure mdmSales = mdmUnitsCube.findOrCreateBaseMeasure("SALES");
NumberSource sales = (NumberSource)mdmSales.getSource();

// Get the Source objects for the PRODUCT_PRIMARY, CHANNEL_PRIMARY
// and the SHIPMENTS hierarchies.
StringSource prodHier = (StringSource)mdmProdHier.getSource();
    StringSource shipHier = (StringSource)mdmShipHier.getSource();
StringSource chanHier = (StringSource)mdmChanHier.getSource();

// Get the YEAR hierarchy level.
List<MdmHierarchyLevel> hierLevels = mdmCalHier.getLevels();
MdmHierarchyLevel mdmYearHierLevel = null;
for(MdmHierarchyLevel mdmHierLevel : hierLevels)
{
  mdmYearHierLevel = mdmHierLevel;
  if(mdmYearHierLevel.getName().equals("YEAR"))
  {
    break;
  }
}
// Get the Source for the Year level of the CALENDAR_YEAR hierarchy.
Source yearLevel = mdmYearHierLevel.getSource();

// Select single values for the hierarchies except for the time hierarchy.
Source prodSel = prodHier.selectValue("PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL");
```

```
Source custSel = shipHier.selectValue("SHIPMENTS::TOTAL_CUSTOMER::TOTAL");
Source chanSel = chanHier.selectValue("CHANNEL_PRIMARY::TOTAL_CHANNEL::TOTAL");

// Get the Long Description attribute for the Time dimension.
MdmBaseAttribute mdmTimeLDAttr = (MdmBaseAttribute)
  mdmTimeDim.getValueDescriptionAttribute();
Source timeLDAttr = mdmTimeLDAttr.getSource();

Source yearsWithLDValue = timeLDAttr.join(yearLevel);

Source result = sales.joinHidden(prodSel)
                     .joinHidden(custSel)
                     .joinHidden(chanSel)
                     .join(yearsWithLDValue);

getContext().commit();
getContext().displayResult(result);
```

The values of the `Cursor` for the `result Source` are the following. The code for formatting the values is not shown. For the complete code for Example 2–4 and Example 2–5, see the `BasicCubeViewQuery.java` example program.

```
Year  Sales
----  ------------
1998  100870876.58
1999  134109248.15
2000  124173521.55
2001  116931722.03
2002   92515295.02
2003  130276513.86
2004  144290685.55
2005  136986571.96
2006  140138317.39
2007  NA
```

3

Discovering Metadata

This chapter describes how to connect to an Oracle Database instance and how to discover existing Oracle OLAP metadata objects. It includes the following topics:

- Connecting to Oracle OLAP
- Overview of the Procedure for Discovering Metadata
- Creating an MdmMetadataProvider
- Getting the MdmSchema Objects
- Getting the Contents of an MdmSchema
- Getting the Objects Contained by an MdmPrimaryDimension
- Getting the Source for a Metadata Object

Connecting to Oracle OLAP

To connect to the Oracle OLAP server in an Oracle Database instance, an OLAP Java API client application uses the Oracle implementation of the Java Database Connectivity (JDBC) API from Sun Microsystems. The Oracle JDBC classes that you use to establish a connection to Oracle OLAP are in the Java archive file `ojdbc5.jar`. For information about getting that file, see Appendix A, "Setting Up the Development Environment".

Prerequisites for Connecting

Before attempting to connect to the Oracle OLAP server, ensure that the following requirements are met:

- The Oracle Database instance is running and was installed with the OLAP option.
- The Oracle Database user ID that you are using for the connection has access to the relational schemas that contain the data.
- The Oracle JDBC and OLAP Java API jar files are in your application development environment. For information about setting up the required jar files, see Appendix A, "Setting Up the Development Environment".

Establishing a Connection

To connect to the OLAP server, perform the following steps:

1. Create a JDBC connection to the database.
2. Create a `DataProvider` and a `UserSession`.

These steps are explained in more detail in the rest of this topic.

Creating a JDBC Connection

One way to create a connection to an Oracle Database instance is to use `oracle.jdbc.OracleDataSource` and `oracle.jdbc.OracleConnection` objects. For example, the following code creates an `oracle.jdbc.OracleDataSource`, sets properties of the object, and then gets a JDBC `OracleConnection` object from the `OracleDataSource`.

The values of the properties for the `OracleDataSource` are from a `Properties` object. The `url` property has the form `jdbc:oracle:thin:@serverName:portNumber:sid`, where `serverName` is the hostname of the server on which the Oracle Database instance is running, `portNumber` is the number of the TCP/IP listener port for the database, and `sid` is the system identifier (SID) of the database instance.

Example 3–1 Getting a JDBC OracleConnection

```
oracle.jdbc.OracleConnection conn = null;
try
{
  OracleDataSource ods = new OracleDataSource();
  ods.setURL(props.getProperty("url"));
  ods.setUser(props.getProperty("user"));
  ods.setPassword(props.getProperty("password"));
  conn = (oracle.jdbc.OracleConnection) ods.getConnection();
}
catch(SQLException e)
{
  System.out.println("Connection attempt failed. " + e);
}
```

In the example, the connection uses the Oracle JDBC thin driver. There are many ways to specify your connection characteristics using the `getConnection` method. There are also other ways to connect to an Oracle Database instance. For more information about Oracle JDBC connections, see *Oracle Database JDBC Developer's Guide*.

After you have the `OracleConnection` object, you can create OLAP Java API `DataProvider` and `UserSession` objects.

Creating a DataProvider and a UserSession

The following code creates a `DataProvider` and a `UserSession`. The `conn` object is the `OracleConnection` from Example 3–1.

Example 3–2 Creating a DataProvider

```
DataProvider dp = new DataProvider();
try
{
  UserSession session = dp.createSession(conn);
}
catch(SQLException e)
{
  System.out.println("Could not create a UserSession. " + e);
}
```

Using the DataProvider, you can get the MdmMetadataProvider, which is described in "Creating an MdmMetadataProvider" on page 3-4. You use the DataProvider to get the TransactionProvider and to create Source and CursorManager objects as described in Chapter 5, "Understanding Source Objects" and Chapter 6, "Making Queries Using Source Methods".

Closing the Connection and the DataProvider

If you are finished using the OLAP Java API, but you want to continue working in your JDBC connection to the database, then use the close method of your DataProvider to release the OLAP Java API resources.

```
dp.close();     // dp is the DataProvider
```

When you have completed your work with the database, use the OracleConnection.close method.

Example 3–3 Closing the Connection

```
try
{
  conn.close();     // conn is the OracleConnection
}
catch(SQLException e)
{
  System.out.println("Cannot close the connection. " + e);
}
```

Overview of the Procedure for Discovering Metadata

The OLAP Java API provides access to the data of an analytic workspace or that is in relational structures. This collection of data is the data store for the application.

Potentially, the data store includes all of the subchemas of the MdmRootSchema. However, the scope of the data store that is visible when an application is running depends on the database privileges that apply to the user ID through which the connection was made. A user can see all of the MdmDatabaseSchema objects that exist under the MdmRootSchema, but the user can see the objects that are owned by an MdmDatabaseSchema only if the user has access rights to the metadata objects. For information on granting access rights and on object security, see *Oracle OLAP User's Guide*.

Purpose of Discovering the Metadata

The metadata objects in the data store help your application to make sense of the data. They provide a way for you to find out what data is available, how it is structured, and what the characteristics of it are.

Therefore, after connecting, your first step is to find out what metadata is available. You can then present choices to the end user about what data to select or calculate and how to display it.

After your application discovers the metadata, it typically goes on to create queries for selecting, calculating, and otherwise manipulating the data. To work with data in these ways, you must get the Source objects from the metadata objects. These Source objects specify the data for querying. For more information on Source objects, see Chapter 5, "Understanding Source Objects".

Steps in Discovering the Metadata

Before investigating the metadata, your application must make a connection to Oracle OLAP. Then, your application might perform the following steps:

1. Create a `DataProvider`.
2. Get the `MdmMetadataProvider` from the `DataProvider`.
3. Get the `MdmRootSchema` from the `MdmMetadataProvider`.
4. Get all of the `MdmDatabaseSchema` objects or get individual ones.
5. Get the `MdmCube`, `MdmDimension`, and `MdmOrganizationalSchema` objects owned by the `MdmDatabaseSchema` objects.

The next four topics in this chapter describe these steps in detail.

Creating an MdmMetadataProvider

An `MdmMetadataProvider` gives access to the metadata in a data store by providing the `MdmRootSchema`. Before you can create an `MdmMetadataProvider`, you must create a `DataProvider` as described in Chapter 4, "Creating Metadata and Analytic Workspaces". Example 3–4 creates an `MdmMetadataProvider`. In the example, `dp` is the `DataProvider`.

Example 3–4 Creating an MdmMetadataProvider

```
MdmMetadataProvider mp = null;
try
{
  mp = (MdmMetadataProvider) dp.getMdmMetadataProvider();
}
catch (Exception e)
{
  println("Cannot get the MDM metadata provider. " + e);
}
```

Getting the MdmSchema Objects

The Oracle OLAP metadata objects that provide access to the data in a data store are organized by `MdmSchema` objects. The top-level `MdmSchema` is the `MdmRootSchema`. Getting the `MdmRootSchema` is the first step in exploring the metadata in your data store. From the `MdmRootSchema`, you can get the `MdmDatabaseSchema` objects. The `MdmRootSchema` has an `MdmDatabaseSchema` for each database user. An `MdmDatabaseSchema` can have `MdmOrganizationalSchema` objects that organize the metadata objects owned by the `MdmDatabaseSchema`.

Example 3–5 demonstrates getting the `MdmRootSchema`, the `MdmDatabaseSchema` objects under it, and any `MdmOrganizationalSchema` objects under them.

Example 3–5 Getting the MdmSchema Objects

```
private void getSchemas(MdmMetadataProvider mp)
{
  MdmRootSchema mdmRootSchema = (MdmRootSchema)mp.getRootSchema();
  List<MdmDatabaseSchema> dbSchemas = mdmRootSchema.getDatabaseSchemas();
  for(MdmDatabaseSchema mdmDBSchema : dbSchemas)
  {
    println(mdmDBSchema.getName());
    getOrgSchemas(mdmDBSchema);
```

```
  }
}

private void getOrgSchemas(MdmSchema mdmSchema)
{
  ArrayList orgSchemaList = new ArrayList();

  if (mdmSchema instanceof MdmDatabaseSchema)
  {
    MdmDatabaseSchema mdmDBSchema = (MdmDatabaseSchema) mdmSchema;
    orgSchemaList = (ArrayList) mdmDBSchema.getOrganizationalSchemas();
  }
  else if (mdmSchema instanceof MdmOrganizationalSchema)
  {
    MdmOrganizationalSchema mdmOrgSchema = (MdmOrganizationalSchema)
                                            mdmSchema;
    orgSchemaList = (ArrayList) mdmOrgSchema.getOrganizationalSchemas();
  }

  if (orgSchemaList.size() > 0)
  {
    println("The MdmOrganizationalSchema subschemas of "
            + mdmSchema.getName() + " are:");
    Iterator orgSchemaListItr = orgSchemaList.iterator();
    while (orgSchemaListItr.hasNext())
    {
      MdmOrganizationalSchema mdmOrgSchema = (MdmOrganizationalSchema)
                                            orgSchemaListItr.next();
      println(mdmOrgSchema.getName());
      getOrgSchemas(mdmOrgSchema);
    }
  }
  else
  {
    println(mdmSchema.getName() + " does not have any" +
            " MdmOrganizationalSchema subschemas.");
  }
}
```

Rather than getting all of the `MdmDatabaseSchema` objects, you can use the `getDatabaseSchema` method of the `MdmRootSchema` to get the schema for an individual user. Example Example 3–6 demonstrates getting the `MdmDatabaseSchema` for the `GLOBAL` user.

Example 3–6 Getting a Single MdmDatabaseSchema

```
MdmDatabaseSchema mdmGlobalSchema = mdmRootSchema.getDatabaseSchema("GLOBAL");
```

Getting the Contents of an MdmSchema

From an `MdmSchema`, you can get all of the subschema, `MdmCube`, `MdmPrimaryDimension`, and `MdmMeasure` objects that it contains. Also, the `MdmRootSchema` has an `MdmMeasureDimension` that has a `List` of all of the available `MdmMeasure` objects.

If you want to display all of the dimensions and methods that are owned by a particular user, then you could get the lists of dimensions and measures from the `MdmDatabaseSchema` for that user. Example 3–7 gets the dimensions and measures from the MdmDatabaseSchema from Example 3–6. It displays the name of each dimension and measure.

Example 3–7 Getting the Dimensions and Measures of an MdmDatabaseSchema

```
private void getObjects(MdmDatabaseSchema mdmGlobalSchema)
{
  List dimList = mdmGlobalSchema.getDimensions();
  String objName = mdmGlobalSchema.getName() + " schema";
  getNames(dimList, "dimensions", objName);

  List measList = mdmGlobalSchema.getMeasures();
  getNames(measList, "measures", objName);
}

private void getNames(List objectList, String objTypes, String objName)
{
  println("The " + objTypes + " of the " + objName + " are:");
  Iterator objListItr = objectList.iterator();
  while (objListItr.hasNext())
  {
    MdmObject mdmObj = (MdmObject) objListItr.next();
    println(mdmObj.getName());
  }
}
```

The output of Example 3–7 is the following.

```
The dimensions of the GLOBAL schema are:
CHANNEL_AWJ
CUSTOMER_AWJ
PRODUCT_AWJ
TIME_AWJ
The measures of the GLOBAL schema are:
UNIT_COST
UNIT_PRICE
UNITS
SALES
```

To display just the dimensions and measures associated with an MdmCube, you could use the getTopLevelObject method of an MdmDatabaseSchema to get the cube and then get the dimensions and measures of the cube. Example 3–8 gets an MdmCube from the MdmDatabaseSchema of Example 3–6 and displays the names of the dimensions and measures associated with it using the getNames method of Example 3–7.

Example 3–8 Getting the Dimensions and Measures of an MdmCube

```
private void getCubeObjects(MdmDatabaseSchema mdmGlobalSchema)
{
  MdmCube mdmUnitsCube = (MdmCube)
                   mdmGlobalSchema.getTopLevelObject("PRICE_CUBE_AWJ");
  String objName = mdmUnitsCube.getName() + " cube";
  List dimList = mdmUnitsCube.getDimensions();
  getNames(dimList, "dimensions", objName);

  List<MdmMeasure> measList = mdmUnitsCube.getMeasures();
  getNames(measList, "measures", objName);
}
```

The output of Example 3–8 is the following.

```
The dimensions of the PRICE_CUBE_AWJ cube are:
TIME_AWJ
PRODUCT_AWJ
The measures of the PRICE_CUBE_AWJ cube are:
UNIT_COST
UNIT_PRICE
```

Getting the Objects Contained by an MdmPrimaryDimension

In discovering the metadata objects to use in creating queries and displaying the data, an application typically gets the `MdmSubDimension` components of an `MdmPrimaryDimension` and the `MdmAttribute` objects that are associated with the dimension. This section demonstrates getting the components and attributes of a dimension.

Getting the Hierarchies and Levels of an MdmPrimaryDimension

An `MdmPrimaryDimension` has one or more component `MdmHierarchy` objects, which you can obtain by calling the `getHierarchies` method of the dimension. That method returns a `List` of `MdmHierarchy` objects. The levels of an `MdmPrimaryDimension` are represented by `MdmDimensionLevel` objects.

If an `MdmHierarchy` is an `MdmLevelHierarchy`, then it has `MdmHierarchyLevel` objects that associate `MdmDimensionLevel` objects with it. You can obtain the `MdmHierarchyLevel` objects by calling the `getHierarchyLevels` method of the `MdmLevelHierarchy`.

Example 3–9 gets an `MdmPrimaryDimension` from the `MdmDatabaseSchema` of Example 3–6 and displays the names of the hierarchies and the levels associated with them.

Example 3–9 Getting the Hierarchies and Levels of a Dimension

```
private void getHierarchiesAndLevels(MdmDatabaseSchema mdmGlobalSchema)
{
  MdmPrimaryDimension mdmCustDim = (MdmPrimaryDimension)
                   mdmGlobalSchema.getTopLevelObject("CUSTOMER_AWJ");
  List<MdmHierarchy> hierList = mdmCustDim.getHierarchies();
  println("The hierarchies of the dimension are:");
  for (MdmHierarchy mdmHier : hierList)
  {
    println(mdmHier.getName());
    if (mdmHier instanceof MdmLevelHierarchy)
    {
      MdmLevelHierarchy mdmLevelHier = (MdmLevelHierarchy) mdmHier;
      List<MdmHierarchyLevel> hierLevelList = mdmLevelHier.getHierarchyLevels();
      println("  The levels of the hierarchy are:");
      for (MdmHierarchyLevel mdmHierLevel : hierLevelList)
      {
        println("  " + mdmHierLevel.getName());
      }
    }
  }
}
```

The output of Example 3–9 is the following.

```
The hierarchies of the dimension are:
SHIPMENTS
  The levels of the hierarchy are:
  TOTAL_CUSTOMER
  REGION
  WAREHOUSE
  SHIP_TO
MARKETS
  The levels of the hierarchy are:
  TOTAL_MARKET
  MARKET_SEGMENT
  ACCOUNT
  SHIP_TO
```

Getting the Attributes for an MdmPrimaryDimension

An `MdmPrimaryDimension` and the hierarchies and levels of it have associated `MdmAttribute` objects. You can obtain many of the attributes by calling the `getAttributes` method of the dimension, hierarchy, or level. That method returns a `List` of `MdmAttribute` objects. You can obtain specific attributes, such as a short or long description attribute or a parent attribute by calling the appropriate method of an `MdmPrimaryDimension` or an `MdmHierarchy`.

Example 3–10 demonstrates getting the `MdmAttribute` objects for an `MdmPrimaryDimension`. It also gets the parent attribute. The example displays the names of the `MdmAttribute` objects.

Example 3–10 Getting the MdmAttribute Objects of an MdmPrimaryDimension

```
private void getAttributes(MdmDatabaseSchema mdmGlobalSchema)
{
  MdmTimeDimension mdmTimeDim = (MdmTimeDimension)
                   mdmGlobalSchema.getTopLevelObject("TIME_AWJ");
  List attrList = mdmTimeDim.getAttributes();
  Iterator attrListItr = attrList.iterator();
  println("The MdmAttribute objects of " + mdmTimeDim.getName() + " are:");
  while (attrListItr.hasNext())
  {
    MdmAttribute mdmAttr = (MdmAttribute) attrListItr.next();
    println("  " + mdmAttr.getName());
  }

  MdmAttribute mdmParentAttr = mdmTimeDim.getParentAttribute();
  println("The parent attribute is " + mdmParentAttr.getName() + ".");
}
```

The output of Example 3–10 is the following.

```
The MdmAttribute objects of TIME_AWJ are:
  LONG_DESCRIPTION
  SHORT_DESCRIPTION
  END_DATE
  TIME_SPAN
The parent attribute is PARENT_ATTRIBUTE.
```

Getting the Source for a Metadata Object

A metadata object represents a set of data, but it does not provide the ability to create queries on that data. The object is informational. It records the existence, structure, and characteristics of the data. It does not give access to the data values.

To access the data values for a metadata object, an application gets the Source object for that metadata object. A Source for a metadata object is a primary Source.

To get the primary Source for a metadata object, an application calls the getSource method of that metadata object. For example, if an application needs to display the quantity of product units sold during the year 1999, then it must use the getSource method of the MdmMeasure for that data, which is mdmUnits in the following example.

Example 3–11 Getting a Primary Source for a Metadata Object

```
Source units = mdmUnits.getSource();
```

For more information about getting and working with primary Source objects, see Chapter 5, "Understanding Source Objects".

4

Creating Metadata and Analytic Workspaces

This chapter describes how to create new metadata objects and map them to relational structures or expressions. It describes how to export and import the definitions of the metadata objects to XML templates. It also describes how to associate the objects with an analytic workspace, and how to build the analytic workspace.

This chapter includes the following topics:

Overview of Creating and Mapping Metadata

The OLAP Java API provides the ability to create persistent metadata objects. The top-level metadata objects exist in the data dictionary of the Oracle Database instance. The API also provides the ability to create transient metadata objects that exist only for the duration of the session. An application can use both types of metadata objects to create queries that retrieve or otherwise use the data in the data store.

Before an OLAP Java API application can create metadata objects, a database administrator must have prepared the Oracle Database instance. The DBA must have set up permanent and temporary tablespaces in the database to support the creation of Oracle OLAP metadata objects and must have granted the privileges that allow the user of the session to create and manage objects. For information on preparing an Oracle Database instance, see *Oracle OLAP User's Guide*.

A dimensional metadata model typically includes the objects described in Chapter 2, "Understanding OLAP Java API Metadata". For detailed information on designing a dimensional metadata model, see *Oracle OLAP User's Guide*.

You implement the dimensional model by creating OLAP Java API metadata objects. You use classes in the `oracle.olapi.metadata.mapping` package to map the metadata objects to relational source objects and to build analytic workspaces. You use classes in the `oracle.olapi.syntax` package to specify `Expression` objects that you use in mapping the metadata. You use classes in the oracle.olapi.metadata.deployment package to deploy the metadata objects in an analytic workspace or in a relational database (ROLAP) organization.

The basic steps for implementing the dimensional model as OLAP Java API objects in an analytic workspace are the following:

1. Create an AW object and MdmPrimaryDimension and MdmCube objects.
2. Deploy the MdmPrimaryDimension and MdmCube objects to the AW.
3. Create MdmDimensionLevel, MdmHierarchy, and MdmAttribute objects for each MdmPrimaryDimension, create MdmHierarchyLevel objects to associate MdmDimensionLevel objects with an MdmHierarchy, and create the MdmMeasure and related objects for the MdmCube objects.
4. Map the metadata objects to the relational sources of the base data.
5. Commit the Transaction, which creates persistent objects in the database.
6. Load data into the objects from the relational sources by building the analytic workspace.

The following sections describe these steps. The examples in this chapter are modified excerpts from the BuildAW11g.java example program, which creates and builds an analytic workspace. The program also exports the analytic workspace to an XML template.

Creating an Analytic Workspace

An analytic workspace is a container for related dimensional objects. It is represented by the AW class in the oracle.olapi.metadata.deployment package. An analytic workspace is owned by an MdmDatabaseSchema.

Example 4–1 demonstrates getting the MdmDatabaseSchema for the GLOBAL user and creating an AW. For an example that gets the MdmRootSchema, see Chapter 3.

Example 4–1 Creating an AW

```
private void createAW(MdmRootSchema mdmRootSchema)
{
  MdmDatabaseSchema mdmDBSchema = mdmRootSchema.getDatabaseSchema("GLOBAL");
  aw = mdmDBSchema.findOrCreateAW("GLOBAL_AWJ");
}
```

Creating the Dimensions, Levels, and Hierarchies

A dimension is a list of unique values that identify and categorize data. Dimensions form the edges of a cube and identify the values in the measures of the cube. A dimension has one or more levels that categorize the dimension members. It can have one or more hierarchies that further categorize the members.

A dimension also has attributes that contain information about dimension members. For descriptions of creating attributes, see "Creating Attributes" on page 4-7.

This section describes how to create objects that represent a dimension and the levels and hierarchies of a dimension.

Creating Dimensions

An OLAP dimension is represented by the MdmPrimaryDimension class. A dimension is owned by an MdmDatabaseSchema. You create a dimension with the findOrCreateTimeDimension or the findOrCreateStandardDimension method of the MdmDatabaseSchema.

Example 4–2 creates a standard dimension that has the name CHANNEL_AWJ. The example creates an AWPrimaryDimensionOrganization object to deploy the dimension in an analytic workspace. The mdmDBSchema and aw objects are created by Example 4–1. The last three lines call the methods of Example 4–3, Example 4–4, and Example 4–9, respectively.

Example 4–2 Creating and Deploying an MdmStandardDimension

```
MdmStandardDimension mdmChanDim =
  mdmDBSchema.findOrCreateStandardDimension("CHANNEL_AWJ");
AWPrimaryDimensionOrganization awChanDimOrg =
  mdmChanDim.findOrCreateAWPrimaryDimensionOrganization(aw);

createAndMapDimensionLevels(mdmChanDim);
createAndMapHierarchies();
commit(mdmChanDim);
```

Creating and Mapping Dimension Levels

An MdmDimensionLevel represents the members of a dimension that are at the same level. Typically, the members of a level are in a column in a dimension table in the relational source. A MemberListMap associates the MdmDimensionLevel with the relational source.

Example 4–3 creates two MdmDimensionLevel objects for the CHANNEL_AWJ dimension and maps the dimension levels to the key columns of the GLOBAL.CHANNEL_DIM table. The example also maps the long description attributes for the dimension levels to columns of that table. The long description attribute, chanLongDescAttr, is created by Example 4–6.

Example 4–3 Creating and Mapping an MdmDimensionLevel

```
private ArrayList<MdmDimensionLevel> dimLevelList = new ArrayList();
private ArrayList<String> dimLevelNames = new ArrayList();
private ArrayList<String> keyColumns = new ArrayList();
private ArrayList<String> lDescColNames = new ArrayList();

private void createAndMapDimensionLevels(MdmPrimaryDimension mdmChanDim)
{
  dimLevelNames.add("TOTAL_CHANNEL");
  dimLevelNames.add("CHANNEL");

  keyColumns.add("GLOBAL.CHANNEL_DIM.TOTAL_ID");
  keyColumns.add("GLOBAL.CHANNEL_DIM.CHANNEL_ID");

  lDescColNames.add("GLOBAL.CHANNEL_DIM.TOTAL_DSC");
  lDescColNames.add("GLOBAL.CHANNEL_DIM.CHANNEL_DSC");

  // Create the MdmDimensionLevel and MemberListMap objects.
  int i = 0;
  for(String dimLevelName : dimLevelNames)
  {
    MdmDimensionLevel mdmDimLevel =
            mdmChanDim.findOrCreateDimensionLevel(dimLevelNames.get(i));
    dimLevelList.add(mdmDimLevel);

    // Create a MemberListMap for the dimension level.
```

```
      MemberListMap mdmDimLevelMemListMap =
                                   mdmDimLevel.findOrCreateMemberListMap();
      ColumnExpression keyColExp = (ColumnExpression)
                              SyntaxObject.fromSyntax(keyColumns.get(i),
                                                      metadataProvider);
      mdmDimLevelMemListMap.setKeyExpression(keyColExp);
      mdmDimLevelMemListMap.setQuery(keyColExp.getQuery());

      // Create an attribute map for the Long Description attribute.
      AttributeMap attrMapLong =
                mdmDimLevelMemListMap.findOrCreateAttributeMap(chanLongDescAttr);

      // Create an expression for the attribute map.
      Expression lDescColExp = (Expression)
                          SyntaxObject.fromSyntax(lDescColNames.get(i),
                                                  metadataProvider);
       attrMapLong.setExpression(lDescColExp);
      i++;
    }
  }
```

Creating and Mapping Hierarchies

An `MdmHierarchy` represents a hierarchy in the dimensional object model. An `MdmHierarchy` can be an instance of the `MdmLevelHierarchy` or the `MdmValueHierarchy` class. An `MdmLevelHierarchy` has an ordered list of `MdmHierarchyLevel` objects that relate `MdmDimensionLevel` objects to the hierarchy.

Creating and Mapping an MdmLevelHierarchy

Example 4–4 creates a hierarchy for the `CHANNEL_AWJ` dimension. It creates hierarchy levels for the hierarchy and associates attributes with the hierarchy levels. It also maps the hierarchy levels and the attributes to relational sources. The example uses the `ArrayList` objects from Example 4–3. It maps the `MdmHierarchyLevel` objects to the same relational source objects as the `MdmDimensionLevel` objects are mapped.

Example 4–4 Creating and Mapping MdmLevelHierarchy and MdmHierarchyLevel Objects

```
private void createAndMapHierarchies()
{
  MdmLevelHierarchy mdmLevelHier =
                 mdmChanDim.findOrCreateLevelHierarchy("CHANNEL_PRIMARY");

  // Create the MdmHierarchyLevel and HierarchyLevelMap objects.
  int i = 0;
  for(String dimLevelName : dimLevelNames)
  {
      MdmDimensionLevel mdmDimLevel =
        mdmChanDim.findOrCreateDimensionLevel(dimLevelName);
      MdmHierarchyLevel mdmHierLevel =
        mdmLevelHier.findOrCreateHierarchyLevel(mdmDimLevel);
      HierarchyLevelMap hierLevelMap =
        mdmHierLevel.findOrCreateHierarchyLevelMap();
     ColumnExpression keyColExp = (ColumnExpression)
                              SyntaxObject.fromSyntax(keyColumns.get(i),
                                                      metadataProvider);
     hierLevelMap.setKeyExpression(keyColExp);
     hierLevelMap.setQuery(keyColExp.getQuery());
```

```
    //Set the MdmDimensionLevel for the MdmHierarchyLevel.
    mdmHierLevel.setDimensionLevel(dimLevelList.get(i));
    i++;
  }
}
```

Creating and Mapping an MdmValueHierarchy

The GLOBAL_AWJ analytic workspace that is used by the examples in this documentation does not have an MdmPrimaryDimension for which an MdmValueHierarchy would be sensible. The sample schema for the user SCOTT does have a table that can serve as an example.

The SCOTT sample schema has a table named emp. That table has columns for employees and for managers. You could create a dimension for employees. You could then create an MdmValueHierarchy in which you map the employee column as the base values for the hierarchy and you map the manager column as the parent relation, as shown in Example 4–5. To be able to create OLAP dimensions, the SCOTT user must be granted the OLAP_USER role and the CREATE SESSION privilege.

In the example, mdmDBSchema is the MdmDatabaseSchema for the SCOTT user, dp is the DataProvider, and mp is the MdmMetadataProvider. The example does not show the code for connecting to the database or getting the DataProvider and creating a UserSession, or getting the MdmMetadataProvider, the MdmRootSchema, or the MdmDatabaseSchema. The code is an excerpt from a class that extends the BaseExample11g example class. That class uses other example class that have methods for committing the current Transaction and for displaying output. For the complete code, see the CreateValueHierarchy.java example program.

Example 4–5 Creating an MdmValueHierarchy

```
// Create an analytic workspace object.
AW aw = mdmDBSchema.findOrCreateAW(awName);
// Create a dimension and deploy it to the analytic workspace.
MdmPrimaryDimension mdmEmpDim =
  mdmDBSchema.findOrCreateStandardDimension("EMP_DIM");
AWPrimaryDimensionOrganization awEmpDimOrg =
  mdmEmpDim.findOrCreateAWPrimaryDimensionOrganization(aw);

// Get the EMP table and the Query for the table.
MdmTable empTable = (MdmTable)mdmDBSchema.getTopLevelObject("EMP");
Query empQuery = empTable.getQuery();

// Create a value hierarchy.
MdmValueHierarchy mdmValHier =
 mdmEmpDim.findOrCreateValueHierarchy("EMPVALHIER");
// Create a map for the hierarchy.
SolvedValueHierarchyMap solvedValHierMap =
 mdmValHier.findOrCreateSolvedValueHierarchyMap();
// Specify the Query, the key expression and the parent key expression for
// the hierarchy.
solvedValHierMap.setQuery(empQuery);
Expression keyExp =
 (Expression)SyntaxObject.fromSyntax("SCOTT.EMP.EMPNO", mp);
solvedValHierMap.setKeyExpression(keyExp);
Expression parentExp =
 (Expression)SyntaxObject.fromSyntax("SCOTT.EMP.MGR", mp);
solvedValHierMap.setParentKeyExpression(parentExp);
```

```
// Create an attribute that relates a name to each dimension member.
MdmBaseAttribute mdmNameAttr =
 mdmEmpDim.findOrCreateBaseAttribute("EMP_NAME");
SQLDataType sdtVC2 = new SQLDataType("VARCHAR2");
mdmNameAttr.setSQLDataType(sdtVC2)
// Create an attribute map for the attribute.
AttributeMap attrMap =
 solvedValHierMap.findOrCreateAttributeMap(mdmNameAttr);
// Create and set an expression for the attribute map.
Expression exp = (Expression)
  SyntaxObject.fromSyntax("SCOTT.EMP.ENAME", mp);
attrMap.setExpression(exp);
mdmValHier.addAttribute(mdmNameAttr);

// Commit the Transaction before building the analytic workspace.
// The getContext method of BaseExample11g returns a Context11g object,
// which has a method that commits the Transaction.
getContext().commit();
BuildItem bldEmpDim = new BuildItem(mdmEmpDim);
ArrayList<BuildItem> items = new ArrayList();
items.add(bldEmpDim);
BuildProcess bldProc = new BuildProcess(items);

// Execute the build.
try
{
 dp.executeBuild(bldProc, 0);
}
catch (Exception ex)
{
 println("Could not execute the BuildProcess.");
 println("Caught: " + ex);
}

//Get the Source objects for the dimension, the hierarchy, and the attribute.
Source empDim = mdmEmpDim.getSource();
Source valHier = mdmValHier.getSource();
Source empNameAttr = mdmNameAttr.getSource();
// Get the parent attribute and get the Source for it.
MdmAttribute mdmParentAttr = mdmEmpDim.getParentAttribute();
Source parentAttr = mdmParentAttr.getSource();

Source parentByEmpByName = parentAttr.join(valHier.join(empNameAttr));
// Sort the values in ascending order by employee number of the managers
// and then by employee number.
Source sortedParentByEmpByName = parentByEmpByName.sortAscending();

// Commit the Transaction before creating a Cursor.
getContext().commit();
// The displayResult method of the Context11g object creates a Cursor and
// displays the results.
println("The managers of the employees are:");
getContext().displayResult(sortedParentByEmpByName);
```

The output of Example 4–5 is the following. It shows the employee name, the employee ID and then the employee ID of the manager. The results are sorted by manager. The employee King does not have a parent and is the highest member of the hierarchy so the manager value for King is NA.

```
The managers of the employees are:
1: ((SCOTT,EMPVALHIER::7788),EMPVALHIER::7566)
2: ((FORD,EMPVALHIER::7902),EMPVALHIER::7566)
3: ((ALLEN,EMPVALHIER::7499),EMPVALHIER::7698)
4: ((WARD,EMPVALHIER::7521),EMPVALHIER::7698)
5: ((MARTIN,EMPVALHIER::7654),EMPVALHIER::7698)
6: ((TURNER,EMPVALHIER::7844),EMPVALHIER::7698)
7: ((JAMES,EMPVALHIER::7900),EMPVALHIER::7698)
8: ((MILLER,EMPVALHIER::7934),EMPVALHIER::7782)
9: ((ADAMS,EMPVALHIER::7876),EMPVALHIER::7788)
10: ((JONES,EMPVALHIER::7566),EMPVALHIER::7839)
11: ((BLAKE,EMPVALHIER::7698),EMPVALHIER::7839)
12: ((CLARK,EMPVALHIER::7782),EMPVALHIER::7839)
13: ((SMITH,EMPVALHIER::7369),EMPVALHIER::7902)
14: ((KING,EMPVALHIER::7839),NA)
```

Creating Attributes

Attributes contain information about dimension members. An `MdmBaseAttribute` represents values that are based on relational source tables. An `MdmDerivedAttribute` represents values that Oracle OLAP derives from characteristics or relationships of the dimension members. For example, the `getParentAttribute()` of an `MdmPrimaryDimension` returns an `MdmDerivedAttribute` that records the parent of each dimension member.

You create a base attribute for a dimension with the `findOrCreateBaseAttribute` method. You specify the data type of the attribute. For some attributes, you make the attribute visible with a method of the dimension like `setValueDescriptionAttribute`.

Example 4–6 creates a long description attribute for the `CHANNEL_AWJ` dimension and makes it visible on the dimension.

Example 4–6 Creating an MdmBaseAttribute

```
private MdmBaseAttribute chanLongDescAttr = null;
private void createLongDesciptionAttribute(MdmPrimaryDimension mdmChanDim)
{
// Create the long description attribute and set the data type for it.
chanLongDescAttr = mdmChanDim.findOrCreateBaseAttribute("LONG_DESCRIPTION");
SQLDataType sdtVC2 = new SQLDataType("VARCHAR2");
chanLongDescAttr.setSQLDataType(sdtVC2);

// Make the attribute visible on the dimension.
mdmChanDim.setValueDescriptionAttribute(chanLongDescAttr);
```

An attribute can have different values for the members of different levels of the dimension. In that case the attribute has an attribute mapping for each level. Example 4–3 creates an `AttributeMap` for the long description attribute for each dimension level by calling the `findOrCreateAttributeMap` method of the `MemberListMap` for each dimension level. It specifies a different column for each attribute map.

Creating Cubes and Measures

A cube in a dimensional object model is represented by the `MdmCube` class. An `MdmCube` owns one or more `MdmMeasure` objects. It has a list of the `MdmPrimaryDimension` objects that dimension the measures.

An MdmCube has the following objects associated with it.

- MdmPrimaryDimension objects that specify the dimensionality of the cube.
- MdmMeasure objects that contain data that is identified by the dimensions.
- A CubeOrganization that specifies how the cube stores and manages the measure data.
- CubeMap objects that associate the cube with relational sources.
- A ConsistentSolveSpecification that specifies how to calculate, or solve, the aggregate level data.

Creating Cubes

This section has an example that creates a cube and some of the objects associated with it. Example 4–7 creates an MdmCube that has the name PRICE_CUBE_AWJ. The example creates an AWCubeOrganization object to deploy the cube in an analytic workspace. The mdmDBSchema and aw objects are created by Example 4–1 and the leafLevel ArrayList is created in Example 4–4. The mdmTimeDim and mdmProdDim objects are dimensions of time periods and product categories. The BuildAW11g program creates those dimensions. The last lines of the example call the methods in Example 4–8 and Example 4–9, respectively.

Example 4–7 Creating and Mapping an MdmCube

```
private MdmCube createAndMapCube(MdmPrimaryDimension mdmTimeDim,
                                 MdmPrimaryDimension mdmProdDim)
{
  MdmCube mdmPriceCube = mdmDBSchema.findOrCreateCube("PRICE_CUBE_AWJ");
  // Add dimensions to the cube.
  mdmPriceCube.addDimension(mdmTimeDim);
  mdmPriceCube.addDimension(mdmProdDim);

  AWCubeOrganization awCubeOrg =
    mdmPriceCube.findOrCreateAWCubeOrganization(aw);
  awCubeOrg.setMVOption(AWCubeOrganization.NONE_MV_OPTION);
  awCubeOrg.setMeasureStorage(AWCubeOrganization.SHARED_MEASURE_STORAGE);
  awCubeOrg.setCubeStorageType("NUMBER");

  AggregationCommand aggCommand = new AggregationCommand("AVG");
  ArrayList<ConsistentSolveCommand> solveCommands = new ArrayList();
  solveCommands.add(aggCommand);
  ConsistentSolveSpecification conSolveSpec =
                      new ConsistentSolveSpecification(solveCommands);
  mdmPriceCube.setConsistentSolveSpecification(conSolveSpec);

  // Create and map the measures of the cube.
  createAndMapMeasures(mdmPriceCube);
  // Commit the Transaction.
  commit(mdmPriceCube);
}
```

Creating and Mapping Measures

This section has an example that creates measures for a cube and maps the measures to fact tables in the relational database. The example uses the cube created by Example 4-6.

Example 4–8 Creating and Mapping Measures

```
private void createAndMapMeasures(MdmCube mdmPriceCube)
{
  ArrayList<MdmBaseMeasure> measures = new ArrayList();
  MdmBaseMeasure mdmCostMeasure =
                             mdmPriceCube.findOrCreateBaseMeasure("UNIT_COST");
  MdmBaseMeasure mdmPriceMeasure =
                            mdmPriceCube.findOrCreateBaseMeasure("UNIT_PRICE");
  SQLDataType sdt = new SQLDataType("NUMBER");
  mdmCostMeasure.setSQLDataType(sdt);
  mdmPriceMeasure.setSQLDataType(sdt);
  measures.add(mdmCostMeasure);
  measures.add(mdmPriceMeasure);
  MdmTable priceCostTable = (MdmTable)
                  mdmDBSchema.getTopLevelObject("PRICE_FACT");
  Query cubeQuery = priceCostTable.getQuery();
  ArrayList<String> measureColumns = new ArrayList();
  measureColumns.add("GLOBAL.PRICE_FACT.UNIT_COST");
  measureColumns.add("GLOBAL.PRICE_FACT.UNIT_PRICE");
  CubeMap cubeMap = mdmPriceCube.findOrCreateCubeMap();
  cubeMap.setQuery(cubeQuery);

  // Create MeasureMap objects for the measures of the cube and
  // set the expressions for the measures. The expressions specify the
  // columns of the fact table for the measures.
  int i = 0;
  for(MdmBaseMeasure mdmBaseMeasure : measures)
  {
    MeasureMap measureMap = cubeMap.findOrCreateMeasureMap(mdmBaseMeasure);
    Expression expr = (Expression)
                   SyntaxObject.fromSyntax(measureColumns.get(i),
                                           metadataProvider);
    measureMap.setExpression(expr);
    i++;
  }

  // Create CubeDimensionalityMap objects for the dimensions of the cube and
  // set the expressions for the dimensions. The expressions specify the
  // columns of the fact table for the dimensions.

  ArrayList<String> factColNames = new ArrayList();
  factColNames.add("GLOBAL.PRICE_FACT.MONTH_ID");
  factColNames.add("GLOBAL.PRICE_FACT.ITEM_ID");
  List<MdmDimensionality> mdmDimltys = mdmPriceCube.getDimensionality();
  for (MdmDimensionality mdmDimlty: mdmDimltys)
  {
    CubeDimensionalityMap cubeDimMap =
                 cubeMap.findOrCreateCubeDimensionalityMap(mdmDimlty);
    MdmPrimaryDimension mdmPrimDim = (MdmPrimaryDimension)
                                  mdmDimlty.getDimension();
    String columnMap = null;
    if (mdmPrimDim.getName().startsWith("TIME"))
    {
      columnMap = factColNames.get(0);
      i = 0;
    }
    else// (mdmPrimDim.getName().startsWith("PRODUCT"))
    {
      columnMap = factColNames.get(1);
      i = 1;
```

```
        }
        Expression expr = (Expression)
                          SyntaxObject.fromSyntax(columnMap,
                                                  metadataProvider);
        cubeDimMap.setExpression(expr);

        // Associate the leaf level of the hierarchy with the cube.
        MdmHierarchy mdmDefHier = mdmPrimDim.getDefaultHierarchy();
        MdmLevelHierarchy mdmLevHier = (MdmLevelHierarchy)mdmDefHier;
        List<MdmHierarchyLevel> levHierList = mdmLevHier.getHierarchyLevels();
        // The last element in the list must be the leaf level of the hierarchy.
        MdmHierarchyLevel leafLevel = levHierList.get(levHierList.size() - 1);
        cubeDimMap.setMappedDimension(leafLevel);
      }
    }
```

Committing Transactions

To save a metadata object as a persistent entity in the database, you must commit the `Transaction` in which you created the object. You can commit a `Transaction` at any time. Committing the `Transaction` after creating a top-level object and the objects that it owns is a good practice.

Example 4–9 gets the `TransactionProvider` from the `DataProvider` for the session and commits the current `Transaction`.

Example 4–9 Committing Transactions

```
private void commit(MdmSource mdmSource)
{
  try
  {
    System.out.println("Committing the transaction for " +
                        mdmSource.getName() + ".");

    (dp.getTransactionProvider()).commitCurrentTransaction();
  }
  catch (Exception ex)
  {
    System.out.println("Could not commit the Transaction. " + ex);
  }
}
```

Exporting and Importing XML Templates

You can save the definition of a metadata object by exporting the object to an XML template. Exporting an object saves the definition of the object and the definitions of any objects that it owns. For example, if you export an `AW` object to XML, then the XML includes the definitions of any `MdmPrimaryDimension` and `MdmCube` objects that the `AW` owns, and the `MdmAttribute`, `MdmMeasure` and other objects owned by the dimensions and cubes.

Example 4–10 exports metadata objects to an XML template and saves it in a file. The code excerpt at the beginning of the example creates a `List` of the objects to export. It adds to the `List` the `aw` object, which is the analytic workspace created by Example 4–1. It then calls the `exportToXML` method.

Example 4–10 Exporting to an XML Template

```
 ... // In some method.
List objectsToExport = new ArrayList();
objectsToExport.add(aw);
exportToXML(objectsToExport, "globalawj.xml");
...
public void exportToXML(List objectsToExport, String fileName)
{
  try
  {
    PrintWriter writer = new PrintWriter(new FileWriter(filename));
    mp.exportFullXML(writer,         // mp is the MdmMetadataProvider
                     objectsToExport,
                     null,           // No Map for renaming objects
                     false);         // Do not include the owner name
    writer.close();
  }
  catch (IOException ie)
  {
    ie.printStackTrace();
  }
}
```

You can import a metadata object definition as an XML template. After importing, you must build the object.

Building an Analytic Workspace

After creating and mapping metadata objects, or importing the XML definition of an object, you must perform the calculations that the objects specify and load the resulting data into physical storage structures.

Example 4–11 creates `BuildItem` objects for the dimensions and cubes of the analytic workspace. It creates a `BuildProcess` that specifies the `BuildItem` objects and passes the `BuildProcess` to the `executeBuild` method of the `DataProvider` for the session.

Example 4–11 Building an Analytic Workspace

```
BuildItem bldChanDim = new BuildItem(mdmChanDim);
BuildItem bldProdDim = new BuildItem(mdmProdDim);
BuildItem bldCustDim = new BuildItem(mdmCustDim);
BuildItem bldTimeDim = new BuildItem(mdmTimeDim);
BuildItem bldUnitsCube = new BuildItem(mdmUnitsCube);
BuildItem bldPriceCube = new BuildItem(mdmPriceCube);
ArrayList<BuildItem> items = new ArrayList();
items.add(bldChanDim);
items.add(bldProdDim);
items.add(bldCustDim);
items.add(bldTimeDim);
items.add(bldUnitsCube);
items.add(bldPriceCube);
BuildProcess bldProc = new BuildProcess(items);
try
{
  dp.executeBuild(bldProc, 0);
}
catch (Exception ex)
{
```

```
    System.out.println("Could not execute the BuildProcess." + ex);
  }
```

5

Understanding Source Objects

This chapter introduces Source objects, which you use to specify a query. With a Source, you specify the data that you want to retrieve from the data store and the analytical or other operations that you want to perform on the data. Chapter 6, "Making Queries Using Source Methods", provides examples of using Source objects. Using Template objects to make modifiable queries is discussed in Chapter 10, "Creating Dynamic Queries".

This chapter includes the following topics:

Overview of Source Objects

After you have used the classes in the oracle.olapi.metadata.mdm package to get MdmSource objects that represent OLAP metadata measures and dimensions, you can get Source objects from them. You can also create other Source objects with methods of a DataProvider. You can then use the Source objects to create a query that specifies the data that you want to retrieve from the database. To retrieve the data, you create a Cursor for the Source.

With the methods of a Source, you can specify selections of dimension or measure values and specify operations on the elements of the Source, such as mathematical calculations, comparisons, and ordering, adding, or removing elements of a query. The Source class has a few basic methods and many shortcut methods that use one or more of the basic methods. The most complex basic methods are the join(Source joined, Source comparison, int comparisonRule, boolean visible) method and the recursiveJoin(Source joined, Source comparison, Source parent, int comparisonRule, boolean parentsFirst, boolean parentsRestrictedToBase, int maxIterations, boolean visible) method. The many other signatures of the join and recursiveJoin methods are shortcuts for certain operations of the basic methods.

In this chapter, the information about the join method applies equally to the recursiveJoin method, except where otherwise noted. With the join method, you can select elements of a Source and, most importantly, you can relate the elements of one Source to those of another Source. For example, to specify the dimension members that retrieving the data of a measure requires, you use a `join` method to relate the dimension to the measure.

A `Source` has certain characteristics, such as a type and a data type, and it sometimes has one or more inputs or outputs. This chapter describes these concepts. It also describes the different kinds of `Source` objects and how you get them, the `join` method and other `Source` methods, and how you use those methods to specify a query.

Kinds of Source Objects

The kinds of `Source` objects that you use to specify data and to perform analysis, and the ways that you get them, are the following:

- Primary `Source` objects, which are returned by the `getSource` method of an `MdmSource` object such as an `MdmDimension` or an `MdmMeasure`. A primary `Source` provides access to the data that the `MdmSource` represents. Getting primary `Source` objects is usually the first step in creating a query. You then typically select elements from the primary `Source` objects, thereby producing derived `Source` objects.
- Derived `Source` objects, which you get by calling some of the methods of a `Source` object. Methods such as `join` return a new `Source` that is based on the `Source` on which you call the method. All queries on the data store, other than a simple list of values specified by the primary `Source` for an `MdmSubDimension`, such as an `MdmLevelHierarchy` or an `MdmLevel`, are derived `Source` objects.
- Fundamental `Source` objects, which are returned by the `getSource` method of a `FundamentalMetadataObject`. These `Source` objects represent the OLAP Java API data types.
- List or range `Source` objects, which are returned by the `createConstantSource`, `createListSource` or `createRangeSource` methods of a `DataProvider`. Typically, you use this kind of `Source` as the `joined` or `comparison` parameter to a `join` method.
- Empty, null, or void `Source` objects. Empty and void `Source` objects are returned by the `getEmptySource` or `getVoidSource` method of a `DataProvider`, and null `Source` objects are returned by the `nullSource` method of a `Source`. An empty `Source` has no elements. A void or null `Source` has one element that has the value of `null`. The difference between them is that the type of a void `Source` is the `FundamentalMetadataObject` for the Value data type, and the type of a null `Source` is the `Source` whose `nullSource` method returned it. Typically, you use these kinds of `Source` objects as the `joined` or `comparison` parameter to a `join` method.
- Dynamic `Source` objects, which are returned by the `getSource` method of a `DynamicDefinition`. A dynamic `Source` is usually a derived `Source`. It is generated by a `Template`, which you use to create a dynamic query that you can revise after interacting with an end user.
- Parameterized `Source` objects, which are returned by the `createSource` methods of a `Parameter`. Like a list or range `Source`, you use a parameterized `Source` as a parameter to the `join` method. Unlike a list or range `Source`, however, you can change the value that the `Parameter` represents after the join operation and thereby change the selection that the derived `Source` represents. You can create a `Cursor` for that derived `Source` and retrieve the results of the query. You can then change the value of the `Parameter`, and, without having to

create a new Cursor for the derived Source, use that same Cursor to retrieve the results of the modified query.

- Placeholder Source objects, which are returned by the getSource method of the FundamentalMetadataObject that represents a placeholder for a specific data type. You get the FundamentalMetadataObject for a placeholder with methods of a FundamentalMetadataProvider such as the getNumberPlaceholder or getStringPlaceholder methods. Oracle OLAP uses placeholder Source objects in Assignment objects in an MdmModel or CustomModel. In an Assignment, a placeholder Source represents the Source for the current dimensioned Source to which the value is being assigned. You can use a placeholder Source in creating a custom dimension member and Oracle OLAP automatically adds an Assignment to the appropriate Model.

The Source class has the following subclasses:

- BooleanSource
- DateSource
- NumberSource
- StringSource

These subclasses have different data types and implement Source methods that require those data types. Each subclass also implements methods unique to it, such as the implies method of a BooleanSource or the indexOf method of a StringSource.

Characteristics of Source Objects

A Source has a data type and a type, a Source identification (ID), and a SourceDefinition. This topic describes these concepts. Some Source objects have one or more inputs or outputs. Those complex concepts are discussed in "Inputs and Outputs of a Source" on page 5-6. Some Source objects have an associated Model object, which is discussed in "Model Objects and Source Objects" on page 5-17.

Data Type of a Source

The FundamentalMetadataObject class represents the data type of the elements of an MdmSource. The data type of a Source is represented by a fundamental Source. For example, a BooleanSource has elements that have Java boolean values. The data type of a BooleanSource is the fundamental Source that represents OLAP Java API Boolean values.

To get the fundamental Source that represents the data type of a Source, call the getDataType method of the Source. You can also get a fundamental Source by calling the getSource method of a FundamentalMetadataObject.

Example 5–1 demonstrates getting the fundamental Source for the OLAP Java API String data type, the Source for the data type of an MdmPrimaryDimension, and the Source for the data type of the Source for the MdmPrimaryDimension, and comparing them to verify that they are all the same object. In the example, dp is the DataProvider and mdmProdDim is the MdmPrimaryDimension for the Product dimension.

Example 5–1 Getting the Data Type of a Source

```
FundamentalMetadataProvider fmp = dp.getFundamentalMetadataProvider();
FundamentalMetadataObject fmoStringDataType = fmp.getStringDataType();
```

```
Source stringDataTypeSource = fmoStringDataType.getSource();
FundamentalMetadataObject fmoMdmProdDimDataType =
                                                mdmProdDim.getDataType();
Source mdmProdDimDataTypeSource = fmoMdmProdDimDataType.getSource();
Source prodDim = mdmProdDim.getSource();
Source prodDimDataTypeSource = prodDim.getDataType();
if(stringDataTypeSource == prodDimDataTypeSource &&
   mdmProdDimDataTypeSource == prodDimDataTypeSource)
  println("The Source objects for the data types are the same.");
else
  println("The Source objects for the data types are not the same.");
```

The example displays the following:

```
The Source objects for the data types are the same.
```

Type of a Source

Along with a data type, a `Source` has a type, which is the `Source` from which the elements of the `Source` are drawn. The type of a `Source` determines whether the `join` method can match the `Source` to an input of another `Source`. The only `Source` that does not have a type is the fundamental `Source` for the OLAP Java API Value data type, which represents the set of all values, and from which all other `Source` objects ultimately descend.

The type of a fundamental `Source` is the data type of the `Source`. The type of a list or range `Source` is the data type of the values of the elements of the list or range `Source`.

The type of a primary `Source` is one of the following:

- The fundamental `Source` that represents the data type of the values of the elements of the primary `Source`. For example, the `Source` returned by `getSource` method of a typical `MdmMeasure` is the fundamental `Source` that represents the set of all OLAP Java API number values.
- The `Source` for the `MdmSource` of which the `MdmSource` of the primary `Source` is a component. For example, the type of the `Source` returned by the `getSource` method of an `MdmLevelHierarchy` is the `Source` for the `MdmPrimaryDimension` of which the hierarchy is a component.

The type of a derived `Source` is one of the following:

- The base `Source`, which is the `Source` whose method returned the derived `Source`. A `Source` returned by the `alias`, `extract`, `join`, `recursiveJoin`, or `value` methods, or one of their shortcuts, has the base `Source` as the type. An exception is the derived `Source` returned by the `distinct` method, whose type is the type of the base `Source` rather than the base `Source` itself.
- A fundamental `Source`. Methods such as `position` and `count` return a `Source` the type of which is the fundamental `Source` for the OLAP Java API Integer data type. Methods that make comparisons, such as `eq`, `le`, and so on, return a `Source` the type of which is the fundamental `Source` for the Boolean data type. Methods that perform aggregate functions, such as the `NumberSource` methods `total` and `average`, return as the type of the `Source` a fundamental `Source` that represents the function.

You can find the type by calling the `getType` method of a `Source`.

A Source derived from another Source is a subtype of the Source from which it is derived. You can use the isSubtypeOf method to determine if a Source is a subtype of another Source.

For example, in Example 5–2 the myList object is a list Source. The example uses myList to select values from prodHier, a Source for the default MdmLevelHierarchy of the MdmPrimaryDimension for the Product dimension. In the example, dp is the DataProvider.

Example 5–2 Using the isSubtypeOf Method

```
Source myList = dp.createListSource(new String[] {
                                     "PRODUCT_PRIMARY::FAMILY::LTPC",
                                     "PRODUCT_PRIMARY::FAMILY::DTPC",
                                     "PRODUCT_PRIMARY::FAMILY::ACC",
                                     "PRODUCT_PRIMARY::FAMILY::MON"});
Source prodSel = prodHier.selectValues(myList);
if (prodSel.isSubtypeOf(prodHier))
  println("prodSel is a subtype of prodHier.");
else
  println("prodSel is not a subtype of prodHier.");
```

Because prodSel is a subtype of prodHier, the condition in the if statement is true and the example displays the following:

```
prodSel is a subtype of prodHier.
```

The type of both myList and prodHier is the fundamental String Source. The type of prodSel is prodHier because the elements of prodSel are derived from the elements of prodHier.

The supertype of a Source is the type of the type of a Source, and so on, up through the types to the Source for the fundamental Value data type. For example, the fundamental Value Source is the type of the fundamental String Source, which is the type of prodHier, which is the type of prodSel. The fundamental Value Source and the fundamental String Source are both supertypes of prodSel. The prodSel Source is a subtype of prodHier, and of the fundamental String Source, and of the fundamental Value Source.

Source Identification and SourceDefinition of a Source

A Source has an identification, an ID, which is a String that uniquely identifies it during the current connection to the database. You can get the identification by calling the getID method of a Source. For example, the following code gets the identification of the Source for the MdmPrimaryDimension for the Product dimension and displays the value.

```
println("The Source ID of prodDim is " + prodDim.getID());
```

The preceding code displays the following:

```
The Source ID of prodDim is Hidden..GLOBAL.PRODUCT_AWJ
```

The text displayed by Example 5–9 has several examples of Source identifications.

Each Source has a SourceDefinition object, which records information about the Source. The different kinds of Source objects have different kinds of SourceDefinition objects. For example, the fundamental Source for an MdmPrimaryDimension has an MdmSourceDefinition, which is a subclass of HiddenDefinition, which is a subclass of SourceDefinition.

The SourceDefinition of a Source that is produced by a call to the join method is an instance of the JoinDefinition class. From a JoinDefinition you can get information about the parameters of the join operation that produced the Source, such as the base Source, the joined Source, the comparison Source, the comparison rule, and the value of the visible parameter.

Inputs and Outputs of a Source

The inputs and the outputs of a Source are complex and powerful aspects of the class. This section describes the concepts of inputs and outputs and provides examples of how they are related.

Inputs of a Source

A Source that has inputs is a **dimensioned** Source. An input of a Source is also a Source. An input indicates that the values of the dimensioned Source depend upon an unspecified set of values of the input. A Source that matches to the input provides the values that the input requires. You match an input to a dimensioned Source by using the join method. For information on how to match a Source to an input, see "Matching a Source To an Input" on page 5-10.

Certain Source objects always have one or more inputs. They are the Source objects for the MdmDimensionedObject subclasses MdmMeasure and MdmAttribute. They have inputs because the values of a measure or attribute are specified by the values of their dimensions. The inputs of the Source for the measure or attribute are the Source objects for the dimensions of the measure or the attribute. Before you can retrieve the data for a measure or an attribute, you must match each input to a Source that provides the required values.

Some Source methods produce a Source that has an input. You can produce a Source that has an input by using the extract, position, or value methods. These methods provide a means of producing a Source whose elements are a subset of the elements of another Source. A Source produced by one of these methods has the base Source as an input.

For example, in the following code, the base Source is prodHier. The value method produces prodHierValues, which has prodHier as an input.

```
Source prodHierValues = prodHier.value();
```

The input provides the means to select values from prodHier, as demonstrated by Example 5–2. The selectValues method in Example 5–2 is a shortcut for the following join method.

```
Source prodSel = prodHier.join(prodHier.value(),
                               myList,
                               Source.COMPARISON_RULE_SELECT,
                               false);
```

The parameters of the join method specify the elements of the base Source that appear in the resulting Source. In the example, the joined parameter is the Source produced by the prodHier.value() method. The resulting unnamed Source has prodHier as an input. The input is matched by the base of the join method, which is also prodHier. The result of the join operation, prodSel, has the values of prodHier that match the values of prodHier that are in the comparison Source, myList.

If the joined Source were prodHier and not the Source produced by prodHier.value(), then the comparison would be between the Source object itself and the values of the comparison Source and not between the values of the Source and the values of the comparison Source. Because the joined Source object does not match any of the values of the comparison Source, the result of the join method would have all of the elements of prodHier instead of having only the values of prodHier that are specified by the values of the joined Source that match the values of the comparison Source as specified by the comparison rule.

The input of a Source produced by the position or value method, and an input intrinsic to an MdmDimensionedObject, are regular inputs. A regular input causes the join method, when it matches a Source to the input, to compare the values of the comparison Source to the values of the Source that has the input rather than to the input Source itself.

The input of a Source produced by the extract method is an extraction input. An extraction input differs from a regular input in that, when a value of the Source that has the extraction input is a Source, the join method extracts the values of the Source that is a value of the Source that has the input. The join method then compares the values of the comparison Source to the extracted values rather than to the Source itself.

A Source can have from zero to many inputs. You can get all of the inputs of a Source by calling the getInputs method, the regular inputs by calling the getRegularInputs method, and the extraction inputs by calling the getExtractionInputs method. Each of those methods returns a Set of Source objects.

Outputs of a Source

The join method returns a Source that has the elements of the base Source that are specified by the parameters of the method. If the value of the visible parameter is true, then the joined Source becomes an output of the returned Source. An output of a Source returned by the join method has the elements of the joined Source that specify the elements of the returned Source. An output is a means of identifying the elements of the joined Source that specify the elements of the Source that has the output.

A Source can have from zero to many outputs. You can get the outputs of a Source by calling the getOutputs method, which returns a List of Source objects.

A Source with more than one output has one or more elements for each set of the elements of the outputs. For example, a Source that represents a measure that has had all of the inputs matched, and has had the Source objects that match the inputs turned into outputs, has a single type element for each set of the elements of the outputs because each data value of the measure is identified by a unique set of the values of the dimensions. A Source that represents dimension values that are selected by some operation performed on the data of a measure, however, might have more than one element for each set of the elements of the outputs. An example is a Source that represents product values that have unit costs greater than a certain amount. Such a Source might have several products for each time period that have a unit cost greater than the specified amount.

Example 5–3 produces a selection of the elements of shipHier, which is a Source for a hierarchy of a dimension of customer values. The customers are grouped by a shipment origination and destination hierarchy.

Example 5–3 Using the join Method To Produce a Source Without an Output

```
Source custValuesToSelect = dp.createListSource(new String[]
                                  {"SHIPMENTS::REGION::AMER",
                                   "SHIPMENTS::REGION::EMEA"});
Source shipHierValues = shipHier.value();
Source custSel = shipHier.join(shipHierValues,
                             custValuesToSelect,
                             Source.COMPARISON_RULE_SELECT,
                             false);
```

The `shipHierValues` `Source` has an input of `shipHier`. In the `join` method in the example, the base `Source`, `shipHier`, matches the input of the joined `Source`, `shipHierValues` because the base and the input are the same object. The `join` method selects the elements of the base `shipHier` whose values match the values of the joined `shipHier` that are specified by the comparison `Source`, `custValuesToSelect`. The method produces a `Source`, `custSel`, that has only the selected elements of `shipHier`. Because the `visible` parameter is `false`, the joined `Source` is not an output of `custSel`. The `custSel` `Source` therefore has only two elements, the values of which are `SHIPMENTS::REGION::AMER` and `SHIPMENTS::REGION::EMEA`.

You produce a `Source` that has an output by specifying `true` as the `visible` parameter to the `join` method. Example 5–4 joins the `Source` objects for the dimension selections from Example 5–2 and Example 5–3 to produce a `Source`, `custSelByProdSel`, that has one output. The `custSelByProdSel` `Source` has the elements from `custSel` that are specified by the elements of `prodSel`.

The comparison `Source` is an empty `Source`, which has no elements and which is the result of the `getEmptySource` method of the `DataProvider`, `dp`. The comparison rule value, `COMPARISON_RULE_REMOVE`, selects only the elements of `prodSel` that are not in the comparison `Source`. Because the comparison `Source` has no elements, all of the elements of the joined `Source` are selected. Each of the elements of the joined `Source` specify all of the elements of the base `Source`. The resulting `Source`, `custSelByProdSel`, therefore has all of the elements of `custSel`.

Because the `visible` parameter is `true` in Example 5–4, `prodSel` is an output of `custSelByProdSel`. Therefore, for each element of the output, `custSelByProdSel` has the elements of `custSel` that are specified by that element of the output. Because the `custSel` and `prodSel` are both simple lists of dimension values, the result is the cross product of the elements of both `Source` objects.

Example 5–4 Using the join Method To Produce a Source With an Output

```
Source custSelByProdSel = custSel.join(prodSel,
                                     dp.getEmptySource(),
                                     Source.COMPARISON_RULE_REMOVE,
                                     true);
```

To actually retrieve the data specified by `custSelByProdSel`, you must create a `Cursor` for it. Such a `Cursor` contains the values shown in the following table, which has headings added that indicate that the values from the output, `prodSel`, are in the left column and the values from the elements of the `custSelByProdSel` `Source`, which are derived from the type, `custSel`, are in the right column.

```
      Output Values                    Type Values
-----------------------------   ------------------------
PRODUCT_PRIMARY::FAMILY::DTPC   SHIPMENTS::REGION::AMER
PRODUCT_PRIMARY::FAMILY::DTPC   SHIPMENTS::REGION::EMEA
PRODUCT_PRIMARY::FAMILY::LTPC   SHIPMENTS::REGION::AMER
```

```
PRODUCT_PRIMARY::FAMILY::LTPC   SHIPMENTS::REGION::EMEA
PRODUCT_PRIMARY::FAMILY::MON    SHIPMENTS::REGION::AMER
PRODUCT_PRIMARY::FAMILY::MON    SHIPMENTS::REGION::EMEA
PRODUCT_PRIMARY::FAMILY::ACC    SHIPMENTS::REGION::AMER
PRODUCT_PRIMARY::FAMILY::ACC    SHIPMENTS::REGION::EMEA
```

The `custSelByProdSel` `Source` has two type elements, and the output of the `custSelByProdSel` has four elements. The number of elements of `custSelByProdSel` is eight because for this `Source`, each output element specifies the same set of two type elements.

Each join operation that specifies a `visible` parameter of `true` adds an output to the list of outputs of the resulting `Source`. For example, if a `Source` has two outputs and you call one of the `join` methods that produces an output, then the `Source` that results from the join operation has three outputs. You can get the outputs of a `Source` by calling the `getOutputs` method, which returns a `List` of `Source` objects.

Example 5–5 demonstrates joining a measure to selections from the dimensions of the measure, thus matching to the inputs of the measure `Source` objects that provide the required elements. Because the last two `join` methods match the dimension selections to the inputs of the measure, the resulting `Source` does not have any inputs. Because the `visible` parameter in those joins is `true`, the last `join` method produces a `Source` that has two outputs.

Example 5–5 gets the `Source` for the measure of unit costs. That `Source`, `unitCost`, has two inputs, which are the primary `Source` objects for the Time and Product dimensions, which are the dimensions of Unit Cost. The example gets the `Source` objects for hierarchies of the dimensions, which are subtypes of the `Source` objects for the dimensions. It produces selections of the hierarchies and then joins those selections to the measure. The result, `unitCostSel`, specifies the unit costs of the selected products at the selected times.

Example 5–5 Using the join Method To Match Source Objects To Inputs

```
Source unitCost = mdmUnitCost.getSource();
Source calendar = mdmCalendar.getSource();
Source prodHier = mdmProdHier.getSource();
Source timeSel = calendar.join(calendar.value(),
                               dp.createListSource(new String[]
                                              {"CALENDAR_YEAR::MONTH::2000.05",
                                               "CALENDAR_YEAR::MONTH::2001.05"}),
                               Source.COMPARISON_RULE_SELECT,
                               false);
Source prodSel = prodHier.join(prodHier.value(),
                               dp.createListSource(new String[]
                                         {"PRODUCT_PRIMARY::ITEM::ENVY STD",
                                          "PRODUCT_PRIMARY::ITEM::ENVY EXE",
                                          "PRODUCT_PRIMARY::ITEM::ENVY ABM"}),
                               Source.COMPARISON_RULE_SELECT,
                               false);
Source unitCostSel = unitCost.join(timeSel,
                                   dp.getEmptySource(),
                                   Source.COMPARISON_RULE_REMOVE,
                                   true)
                             .join(prodSel,
                                   dp.getEmptySource(),
                                   Source.COMPARISON_RULE_REMOVE,
                                   true);
```

The unnamed Source that results from joining timeSel to unitCost has one output, which is timeSel. Joining prodSel to that unnamed Source produces unitCostSel, which has two outputs, timeSel and prodSel. The unitCostSel Source has the elements from the type, unitCost, that are specified by the outputs.

A Cursor for unitCostSel contains the following, displayed as a table with headings added that indicate the structure of the Cursor. A Cursor has the same structure as the associated Source. The unit cost values are formatted as dollar values.

```
         Output 1                      Output 2               Type
          Values                        Values               Values
-------------------------------  ----------------------------  --------
PRODUCT_PRIMARY::ITEM::ENVY ABM  CALENDAR_YEAR::MONTH::2000.05  2847.47
PRODUCT_PRIMARY::ITEM::ENVY ABM  CALENDAR_YEAR::MONTH::2001.05  2819.85
PRODUCT_PRIMARY::ITEM::ENVY STD  CALENDAR_YEAR::MONTH::2000.05  2897.40
PRODUCT_PRIMARY::ITEM::ENVY STD  CALENDAR_YEAR::MONTH::2001.05  2376.73
PRODUCT_PRIMARY::ITEM::ENVY EXE  CALENDAR_YEAR::MONTH::2000.05  3238.36
PRODUCT_PRIMARY::ITEM::ENVY EXE  CALENDAR_YEAR::MONTH::2001.05  3015.90
```

Output 1 has the values from prodSel, output 2 has the values from timeSel, and the type values are the values from unitCost that are specified by the output values.

Because these join operations are performed by most OLAP Java API applications, the API provides shortcuts for these and many other join operations. Example 5–6 uses shortcuts for the join operations in Example 5–5 to produce the same result.

Example 5–6 Using Shortcuts

```
Source unitCost = mdmUnitCost.getSource();
StringSource calendar = (StringSource) mdmCalendar.getSource();
StringSource prodHier =(StringSource) mdmProdHier.getSource();
Source timeSel = calendar.selectValues(new String[]
                                       {"CALENDAR_YEAR::MONTH::2000.05",
                                        "CALENDAR_YEAR::MONTH::2001.05"}),
Source prodSel = prodHier.selectValues(new String[]
                                        {"PRODUCT_PRIMARY::ITEM::ENVY STD",
                                         "PRODUCT_PRIMARY::ITEM::ENVY EXE",
                                         "PRODUCT_PRIMARY::ITEM::ENVY ABM"}),
Source unitCostSel = unitCost.join(timeSel).join(prodSel);
```

Matching a Source To an Input

In a join operation, a Source-to-input match occurs only between the base Source and the joined Source. A Source matches an input if one of the following conditions is true.

1. The Source is the same object as the input or it is a subtype of the input.
2. The Source has an output that is the same object as the input or the output is a subtype of the input.
3. The output has an output that is the same object as the input or is a subtype of the input.

The join operation looks for the conditions in the order in the preceding list. It searches the list of outputs of the Source recursively, looking for a match to the input. The search ends with the first matching Source. An input can match with only one Source, and two inputs cannot match with the same Source.

When a Source matches an input, the result of the join method has the elements of the base that match the elements specified by the parameters of the method. You can determine if a Source matches another Source, or an output of the other Source, by passing the Source to the findMatchFor method of the other Source.

When a Source matches an input, the resulting Source does not have that input. Matching a Source to an input does not affect the outputs of the base Source or the joined Source. If a base Source has an output that matches the input of the joined Source, the resulting Source does not have the input but it does have the output.

If the base Source or the joined Source in a join operation has an input that is not matched in the operation, then the unmatched input is an input of the resulting Source.

The comparison Source of a join method does not participate in the input matching. If the comparison Source has an input, then that input is not matched and the Source returned by the join method has that same input.

Example 5–7 demonstrates a base Source matching the input of the joined Source in a join operation. The example uses the position method to produce a Source that has an input, and then uses the join method to match the base of the join operation to the input of the joined Source.

Example 5–7 Matching the Base Source to an Input of the Joined Source

```
Source myList = dp.createListSource(new String[]
                                    {"PRODUCT_PRIMARY::FAMILY::LTPC",
                                     "PRODUCT_PRIMARY::FAMILY::DTPC",
                                     "PRODUCT_PRIMARY::FAMILY::ACC",
                                     "PRODUCT_PRIMARY::FAMILY::MON"});
Source pos = dp.createListSource(new int[] {2, 4});
Source myListPos = myList.position();
Source myListSel = myList.join(myListPos,
                               pos,
                               Source.COMPARISON_RULE_SELECT,
                               false);
```

In Example 5–7, the position method returns myListPos, which has the elements of myList and which has myList as an input. The join method matches the base myList to the input of the joined Source, myListPos.

The comparison Source, pos, specifies the positions of the elements of myListPos to match to the positions of the elements of myList. The elements of the resulting Source, myListSel, are the elements of myList whose positions match those specified by the parameters of the join method.

A Cursor for myListSel has the following values.

```
PRODUCT_PRIMARY::FAMILY::DTPC
PRODUCT_PRIMARY::FAMILY::MON
```

If the visible parameter in Example 5–7 were true instead of false, then the result would have elements from myList and an output of myListPos. A Cursor for myListSel in that case would have the following values, displayed as a table with headings added that indicate the output and type values.

```
Output           Type
Values          Values
------  -------------------------
  2     PRODUCT_PRIMARY::FAMILY::DTPC
  4     PRODUCT_PRIMARY::FAMILY::MON
```

Example 5–8 demonstrates matching outputs of the joined Source to two inputs of the base Source. In the example, units is a Source for an MdmMeasure. It has as inputs the primary Source objects for the Time, Product, Customer, and Channel dimensions.

The DataProvider is dp, and prodHier, shipHier, calendar, and chanHier are the Source objects for the default hierarchies of the Product, Customer, Time, and Channel dimensions, respectively. Those Source objects are subtypes of the Source objects for the dimensions that are the inputs of units.

The join method of prodHier in the first line of Example 5–8 results in prodSel, which specifies selected product values. In that method, the joined Source is the result of the value method of prodHier. The joined Source has the same elements as prodHier, and it has prodHier as an input. The comparison Source is the list Source that is the result of the createListSource method of the DataProvider.

The base Source of the join method, prodHier, matches the input of the joined Source. Because prodHier is the input of the joined Source, the Source returned by the join method has only the elements of the base, prodHier, that match the elements of the joined Source that appear in the comparison Source. Because the visible parameter value is false, the resulting Source does not have the joined Source as an output. The next three similar join operations in Example 5–8 result in selections for the other three dimensions.

The join method of timeSel has custSel as the joined Source. The comparison Source is the result of the getEmptySource method, so it has no elements. The comparison rule specifies that the elements of the joined Source that are present in the comparison Source do not appear in the resulting Source. Because the comparison Source has no elements, all of the elements of the joined Source are selected. The true value for the visible parameter causes the joined Source to be an output of the Source returned by the join method. The returned Source, custSelByTime, has the selected elements of the Customer dimension and has timeSel as an output.

The join method of prodSel has custSelByTime as the joined Source. It produces prodByCustByTime, which has the selected elements from the Product dimension and has custSelByTime as an output. Example 5–8 then joins the dimension selections to the units Source.

The dimension selections are subtypes of the Source objects that are the inputs of units, and therefore the selections match the inputs of units. The input for the Product dimension is matched by prodByCustByTime because prodByCustByTime is a subtype of prodSel, which is a subtype of prodHier. The input for the Customer dimension is matched by the custSelByTime, which is the output of prodByCustByTime.

The custSelByTime Source is a subtype of custSel, which is a subtype of shipHier. The input for the times dimension is matched by timeSel, which is the output of custSelByTime. The timeSel Source is a subtype of calendar.

Example 5–8 Matching an Input of the Base Source to an Output of the Joined Source

```
Source prodSel = prodHier.join(prodHier.value(),
                               dp.createListSource(new String[]
                                     {"PRODUCT_PRIMARY::FAMILY::DTPC",
                                      "PRODUCT_PRIMARY::FAMILY::LTPC"}),
                               Source.COMPARISON_RULE_SELECT,
                               false);
```

```
Source custSel = shipHier.join(shipHier.value(),
                               dp.createListSource(new String[]
                                    {"SHIPMENTS::REGION::EMEA",
                                     "SHIPMENTS::REGION::AMER"}),
                               Source.COMPARISON_RULE_SELECT,
                               false);
Source timeSel =  calendar.join(calendar.value(),
                                dp.createConstantSource(
                                               "CALENDAR_YEAR::YEAR::CY2001"),
                                Source.COMPARISON_RULE_SELECT,
                                false);
Source chanSel = chanHier.join(chanHier.value(),
                                dp.createConstantSource(
                                      "CHANNEL_PRIMARY::CHANNEL::INT"),
                                Source.COMPARISON_RULE_SELECT,
                                false);

Source custSelByTime = custSel.join(timeSel,
                                    dp.getEmptySource(),
                                    Source.COMPARISON_RULE_REMOVE,
                                    true);
Source prodByCustByTime = prodSel.join(custSelByTime,
                                       dp.getEmptySource(),
                                       Source.COMPARISON_RULE_REMOVE,
                                       true);

Source selectedUnits = units.join(prodByCustByTime,
                                  dp.getEmptySource(),
                                  Source.COMPARISON_RULE_REMOVE,
                                  true)
                            .join(promoSel,
                                  dp.getEmptySource(),
                                  Source.COMPARISON_RULE_REMOVE,
                                  true ),
                            .join(chanSel,
                                  dp.getEmptySource(),
                                  Source.COMPARISON_RULE_REMOVE,
                                  true);
```

A `Cursor` for `selectedUnits` contains the following values, displayed in a crosstab format with column headings and formatting added. The table has only the local values of the dimension elements. The first two lines are the page edge values of the crosstab, which are the values of the `chanSel` output of `selectedUnits`, and the value of `timeSel`, which is an output of the `prodByCustByTime` output of `selectedUnits`. The row edge values of the crosstab are the customer values in the left column, and the column edge values are the products values that head the middle and right columns.

The crosstab has only the local value portion of the unique values of the dimension elements. The measure values are the units sold values specified by the selected dimension values.

```
INT
CY2001
             Products
            -----------
Customers   DTPC   LTPC
---------   ----   ----
AMER        1748   846
EMEA         439   215
```

The following table has the same results except that the dimension element values are replaced by the short descriptions of those values.

```
Internet
2001
                           Products
               -------------------------
Customers      Desktop PCs   Portable PCs
-------------  -----------   ------------
North America      1748           846
Europe              439           215
```

To demonstrate turning inputs into outputs, Example 5–9 uses `units`, which is the `Source` for the Units measure, and `defaultHiers`, which is an `ArrayList` of the `Source` objects for the default hierarchies of the dimensions of the measure. The example gets the inputs and outputs of the `Source` for the measure. It displays the `Source` identifications of the `Source` for the measure and for the inputs of the `Source`. The inputs of the `Source` for the measure are the `Source` objects for the `MdmPrimaryDimension` objects that are the dimensions of the measure.

Example 5–9 next displays the number of inputs and outputs of the `Source` for the measure. Using the `join(Source joined)` method, which produces a `Source` that has the elements of the base of the join operation as the elements of it and the `joined` parameter `Source` as an output, it joins one of the hierarchy `Source` objects to the `Source` for the measure, and displays the number of inputs and outputs of the resulting `Source`. It then joins each remaining hierarchy `Source` to the result of the previous join operation and displays the number of inputs and outputs of the resulting `Source`.

Finally the example gets the outputs of the `Source` produced by the last join operation, and displays the `Source` identifications of the outputs. The outputs of the last `Source` are the `Source` objects for the default hierarchies, which the example joined to the `Source` for the measure. Because the `Source` objects for the hierarchies are subtypes of the `Source` objects for the `MdmPrimaryDimension` objects that are the inputs of the measure, they match those inputs.

Example 5–9 Matching the Inputs of a Measure and Producing Outputs

```
Set inputs = units.getInputs();
Iterator inputsItr = inputs.iterator();
List outputs = units.getOutputs();
Source input = null;

int i = 1;
println("The inputs of " + units.getID() + " are:");
while(inputsItr.hasNext())
{
  input = (Source) inputsItr.next();
  println(i + ": " + input.getID());
  i++;
}

println(" ");
int setSize = inputs.size();
for(i = 0; i < (setSize + 1); i++)
{
  println(units.getID() + " has " + inputs.size() +
                 " inputs and " + outputs.size() + " outputs.");
  if (i < setSize)
```

```
  {
    input = defaultHiers.get(i);
    println("Joining " + input.getID() + " to " + units.getID());
    units = units.join(input);
    inputs = units.getInputs();
    outputs = units.getOutputs();
  }
}

println("The outputs of " + units.getID() + " are:");
Iterator outputsItr = outputs.iterator();
i = 1;
while(outputsItr.hasNext())
{
  Source output = (Source) outputsItr.next();
  println(i + ": " + output.getID());
  i++;
}
```

The text displayed by the example is the following:

```
The inputs of Hidden..GLOBAL.UNITS_CUBE_AWJ.UNITS are:
1: Hidden..GLOBAL.CHANNEL_AWJ
2: Hidden..GLOBAL.CUSTOMER_AWJ
3: Hidden..GLOBAL.PRODUCT_AWJ
4: Hidden..GLOBAL.TIME_AWJ

Hidden..GLOBAL.UNITS_CUBE_AWJ.UNITS has 4 inputs and 0 outputs.
Joining Hidden..GLOBAL.CHANNEL_AWJ.CHANNEL_PRIMARY to Hidden..GLOBAL.UNITS_CUBE_
AWJ.UNITS
Join.30 has 3 inputs and 1 outputs.
Joining Hidden..GLOBAL.CUSTOMER_AWJ.SHIPMENTS to Join.30
Join.31 has 2 inputs and 2 outputs.
Joining Hidden..GLOBAL.PRODUCT_AWJ.PRODUCT_PRIMARY to Join.31
Join.32 has 1 inputs and 3 outputs.
Joining Hidden..GLOBAL.TIME_AWJ.CALENDAR_YEAR to Join.32
Join.33 has 0 inputs and 4 outputs.

The outputs of Join.33 are:
1: Hidden..GLOBAL.TIME_AWJ.CALENDAR_YEAR
2: Hidden..GLOBAL.PRODUCT_AWJ.PRODUCT_PRIMARY
3: Hidden..GLOBAL.CUSTOMER_AWJ.SHIPMENTS
4: Hidden..GLOBAL.CHANNEL_AWJ.CHANNEL_PRIMARY
```

Note that as each successive `Source` for a hierarchy is joined to the result of the previous join operation, it becomes the first output in the `List` of outputs of the resulting `Source`. Therefore, the first output of `Join.33` is `Hidden..GLOBAL.TIME_AWJ.CALENDAR_YEAR`, and the last output is `Hidden..GLOBAL.CHANNEL_AWJ.CHANNEL_PRIMARY`.

Describing Parameterized Source Objects

Parameterized `Source` objects provide a way of specifying a query and retrieving different result sets for the query by changing the set of elements specified by the parameterized `Source`. You create a parameterized `Source` with a `createSource` method of the `Parameter`. The `Parameter` supplies the value that the parameterized `Source` specifies.

A typical use of a Parameter object is to specify the page edges of a cube. Example 6–9 demonstrates using Parameter objects to specify page edges. Another use of a Parameter is to fetch from the server only the set of elements that you currently need. Example 6–15 demonstrates using Parameter objects to fetch different sets of elements.

When you create a Parameter object, you supply an initial value for the Parameter. You then create the parameterized Source using the Parameter. You include the parameterized Source in specifying a query. You create a Cursor for the query. You can change the value of the Parameter with the setValue method, which changes the set of elements that the query specifies. Using the same Cursor, you can then display the new set of values.

Example 5–10 demonstrates the use of a Parameter and a parameterized Source to specify an element in a measure dimension. It creates a list Source that has as element values the Source objects for the Unit Cost and Unit Price measures. The example creates a StringParameter object that has as an initial value the unique identifying String for the Source for the Unit Cost measure. That StringParameter is then used to create a parameterized Source.

The example extracts the values from the measures, and then selects the data values that are specified by joining the dimension selections to the measure specified by the parameterized Source. It creates a Cursor for the resulting query and displays the results. After resetting the Cursor position and changing the value of the measParam StringParameter, the example displays the values of the Cursor again.

The dp object is the DataProvider. The getContext method gets a Context11g object that has a method that displays the values of the Cursor with only the local value of the dimension elements.

Example 5–10 Using a Parameterized Source With a Measure Dimension

```
Source measDim = dp.createListSource(new Source[] {unitCost,
                                                   unitPrice});

// Get the unique identifiers of the Source objects for the measures.
String unitCostID = unitCost.getID();
String unitPriceID = unitPrice.getID();

// Create a StringParameter using one of the IDs as the initial value.
StringParameter measParam = new StringParameter(dp, unitCostID);

// Create a parameterized Source.
Source measParamSrc = measParam.createSource();

// Extract the values from the measure dimension elements, and join
// them to the specified measure and the dimension selections.
Source result = measDim.extract().join(measDim, measParamSrc)
                                 .join(prodSelShortDescr)
                                 .join(timeSelShortDescr);
// Get the TransactionProvider and commit the current transaction.
// These operations are not shown.

// Create a Cursor.
CursorManager cursorMngr = getDataProvider().createCursorManager(result);
Cursor resultsCursor = cursorMngr.createCursor();

// Display the results.
getContext().displayCursor(resultsCursor, true);
```

```
// Reset the Cursor position to 1.
resultsCursor.setPosition(1);

// Change the value of the parameterized Source.
measParam.setValue(unitPriceID);

// Display the results again.
getContext().displayCursor(resultsCursor, true);
```

The following table displays the first set of values of `resultsCursor`, with column headings and formatting added. The left column of the table has the local value of the Time dimension hierarchy. The second column from the left has the short value description of the time value. The third column has the local value of the Product dimension hierarchy. The fourth column has the short value description of the product value. The fifth column has the Unit Cost measure value for the time and product.

```
Time     Description  Product     Description     Unit Cost
-------  -----------  --------  ---------------  ---------
2001.04     Apr-01    ENVY EXE  Envoy Executive    2952.85
2001.04     Apr-01    ENVY STD  Envoy Standard     2360.78
2001.05     May-01    ENVY EXE  Envoy Executive    3015.90
2001.05     May-01    ENVY STD  Envoy Standard     2376.73
```

The following table displays the second set of values of `resultsCursor` in the same format. This time the fifth column has values from the Unit Price measure.

```
Time     Description  Product     Description     Unit Price
-------  -----------  --------  ---------------  ----------
2001.04     Apr-01    ENVY EXE  Envoy Executive     3107.65
2001.04     Apr-01    ENVY STD  Envoy Standard      2412.42
2001.05     May-01    ENVY EXE  Envoy Executive     3147.85
2001.05     May-01    ENVY STD  Envoy Standard      2395.63
```

Model Objects and Source Objects

This topic describes the `Model` interface and the implementations of it, and the relationship of `Model` and `Source` objects. It also presents examples of creating custom `Model` objects and performing other tasks that involve `Source` and `Model` objects.

Describing the Model for a Source

A `Model` is analogous to the Oracle SQL `MODEL` clause. With a `Model` you can assign a value to the `Source` for a dimensioned object for one or more sets of members of the dimensions of the object. The value that the `Model` assigns can be anything from a simple constant to the result of a complex calculation involving several other `Source` objects with nested `Model` objects.

The value that a `Model` assigns for a set of dimension members is represented by an `Assignment` object. A `Model` can have one or more `Assignment` objects. Each dimension member in the set is represented by a `Qualification` object. An `Assignment` has one or more `Qualification` objects.

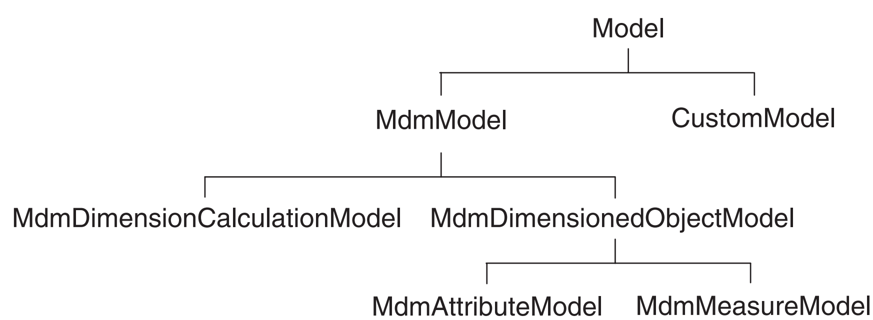
The value that the `Assignment` assigns is specified by a `Source`. An `Assignment` also has an integer that specifies a precedence that affects the order in which Oracle OLAP calculates a value and assigns it. If you create more that one `Assignment` for a `Model` without specifying a precedence, then the order in which Oracle OLAP calculates and assigns the values is not guaranteed.

A Model assigns values for existing dimension members. You can use a Model to assign a different value for a dimension member, or to assign a value for a set of members of more than one dimension, or to assign a different value for a specific measure for the set of dimension members, or to assign a value for the dimension member for an attribute.

When you create a custom dimension member, you specify an assignment value for it. Oracle OLAP automatically adds an Assignment object that specifies the value for the custom member to the appropriate Model for the dimension. Oracle OLAP assigns that value as the measure value for any measure dimensioned by the dimension.

Figure 5–1 illustrates the class hierarchy of the Model interface and the classes that implement it. The oracle.olapi.metadata.mdm.MdmModel class implements the Model interface for MdmObject objects. Another implementation of the Model interface is the CustomModel class in the oracle.olapi.data.source package.

Figure 5–1 The Model Interface and Implementations



A Model has one or more inputs, which are the Source objects for which the model assigns values. The inputs are equivalent to the list of dimensions of an OLAP DML or SQL Model. For example, the MdmDimensionCalculationModel returned by the getNumberCalcModel method of an MdmStandardDimension has as an input the Source for that same MdmStandardDimension. The MdmDimensionedObjectModel returned by the getModel method of an MdmAttribute has as an input the Source for the MdmPrimaryDimension that dimensions the attribute. The MdmDimensionedObjectModel returned by getModel method of an MdmMeasure has as inputs the Source objects for the MdmPrimaryDimension objects that dimension the measure.

A Model can have one or more parents, which are other Model objects from which the Model inherits Assignment objects. An MdmMeasureModel has as parents the MdmDimensionCalculationModel objects of the dimensions associated with it. MdmAttributeModel and MdmDimensionCalculationModel objects do not have parent Model objects.

A CustomModel can have inputs and it can have parent Model objects. When you create a CustomModel object, you can specify inputs and parent Model objects for it. A CustomModel can have also have outputs, which MdmModel objects do not have.

You can create a series of CustomModel objects and have them inherit Assignment objects from each other. The following restrictions apply to the inheritance of an Assignment by one CustomModel from another:

- The inheritance cannot be circular. For example, if customModelB inherits from customModelA, then customModelA cannot inherit from customModelB.
- The type and the outputs of the CustomModel objects must be the same.

- If a parent `CustomModel` has an input, then the child `CustomModel` must also specify that input. The child `CustomModel` can have additional inputs, but it must specify the inputs of the parent `CustomModel` objects.

After creating a `CustomModel` and adding any assignments to it, you can create a `Source` for it by calling the `createSolvedSource` method of the `CustomModel`. With the `defaultValues` parameter of the `createSolvedSource` method, you can specify a `Source` that supplies default values for the `Source` returned by the method. If you do not specify a `Source` for the default values, then the default values of the resulting `Source` are `null`.

Creating a CustomModel - Example

The `Source.extract` method is implemented as a `CustomModel`. An advantage of using your own `CustomModel` over the `extract` method is that you can assign the measure value to a `String` other than a `Source` ID. Example 5–11 demonstrates using the `extract` method and then using a `CustomModel` to achieve the same result. It also demonstrates using another `CustomModel` to achieve a result that assigns the measure values to a different set of `String` values.

In the example, `unitPrice` and `unitCost` are `NumberSource` objects for the Unit Price and Unit Cost measures, and `dp` is the `DataProvider`. The `prodSel` object is a `Source` that represents the selection of three members of the Product dimension.

Example 5–11 Implementing the extract Method As a CustomModel

```
// Create a Source that represents a calculation involving two measures.
Source calculation = unitPrice.minus(unitCost);

// Create a list Source that has Source objects as element values.
Source sourceListSrc = dp.createListSource(new Source[]
                                           {unitPrice, unitCost, calculation});
// Use the extract method to get the values of the Source components of the
// list and join Source objects that match the inputs.
Source resultUsingExtract =
                   sourceListSrc.extract()
                                .join(sourceListSrc)
                                .join(prodSel)
                                .join(calendar, "CALENDAR_YEAR::MONTH::2000.05");

// Produce the same result using a CustomModel directly.
CustomModel  customModel = dp.createModel(sourceListSrc);
customModel.assign(unitPrice.getID(), unitPrice);
customModel.assign(unitCost.getID(), unitCost);
customModel.assign(calculation.getID(), calculation);
Source measValForSrc = customModel.createSolvedSource();
Source resultUsingCustomModel =
                   measValForSrc.join(sourceListSrc)
                                .join(prodSel)
                                .join(calendar, "CALENDAR_YEAR::MONTH::2000.05");

// Create a list Source that has String objects as element values.
Source stringListSrc = dp.createListSource(new String[]
                                           {"price", "cost", "markup"});
// Create a CustomModel for the list Source.
CustomModel  customModel2 = dp.createModel(stringListSrc);
customModel2.assign("price", unitPrice);
customModel2.assign("cost", unitCost);
customModel2.assign("markup", calculation);
```

```
Source measValForSrc2 = customModel2.createSolvedSource();

Source resultUsingCustomModel2 =
                  measValForSrc2.join(stringListSrc)
                                .join(prodSel)
                                .join(calendar, "CALENDAR_YEAR::MONTH::2000.05");
```

`Cursor` objects for `resultUsingExtract` and `resultUsingCustomModel` have the same values, which are the following, shown with formatting added:

```
PRODUCT_PRIMARY::ITEM::ENVY ABM  Hidden..GLOBAL.PRICE_CUBE_AWJ.UNIT_PRICE
 2962.14
PRODUCT_PRIMARY::ITEM::ENVY ABM  Hidden..GLOBAL.PRICE_CUBE_AWJ.UNIT_COST
 2847.47
PRODUCT_PRIMARY::ITEM::ENVY ABM  Join.2
 114.67
PRODUCT_PRIMARY::ITEM::ENVY EXE  Hidden..GLOBAL.PRICE_CUBE_AWJ.UNIT_PRICE
 3442.86
PRODUCT_PRIMARY::ITEM::ENVY EXE  Hidden..GLOBAL.PRICE_CUBE_AWJ.UNIT_COST
 3238.36
PRODUCT_PRIMARY::ITEM::ENVY EXE  Join.2
 204.50
ITEM::ENVY STD  Hidden..GLOBAL.PRICE_CUBE_AWJ.UNIT_PRICE
 3118.61
PRODUCT_PRIMARY::ITEM::ENVY STD  Hidden..GLOBAL.PRICE_CUBE_AWJ.UNIT_COST
 2897.40
PRODUCT_PRIMARY::ITEM::ENVY STD  Join.2
 221.21
```

A `Cursor` for `resultUsingCustomModel2` has the following values, shown with formatting added:

```
PRODUCT_PRIMARY::ITEM::ENVY ABM  price  2962.14
PRODUCT_PRIMARY::ITEM::ENVY ABM  cost   2847.47
PRODUCT_PRIMARY::ITEM::ENVY ABM  markup  114.67
PRODUCT_PRIMARY::ITEM::ENVY EXE  price  3442.86
PRODUCT_PRIMARY::ITEM::ENVY EXE  cost   3238.36
PRODUCT_PRIMARY::ITEM::ENVY EXE  markup  204.50
PRODUCT_PRIMARY::ITEM::ENVY STD  price  3118.61
PRODUCT_PRIMARY::ITEM::ENVY STD  cost   2897.40
PRODUCT_PRIMARY::ITEM::ENVY STD  markup  221.21
```

6

Making Queries Using Source Methods

You create a query by producing a Source that specifies the data that you want to retrieve from the data store and any operations that you want to perform on that data. To produce the query, you begin with the primary Source objects that represent the metadata of the measures and the dimensions and their attributes that you want to query. Typically, you use the methods of the primary Source objects to derive a number of other Source objects, each of which specifies a part of the query, such as a selection of dimension elements or an operation to perform on the data. You then join the primary and derived Source objects that specify the data and the operations that you want. The result is one Source that represents the query.

This chapter briefly describes the various kinds of Source methods, and discusses some of them in greater detail. It also discusses how to make some typical OLAP queries using these methods and provides examples of some of them.

This chapter includes the following topics:

- Describing the Basic Source Methods
- Using the Basic Methods
- Using Other Source Methods

Describing the Basic Source Methods

The Source class has many methods that return a derived Source. The elements of the derived Source result from operations on the base Source, which is the Source whose method is called that produces the derived Source. Only a few methods perform the most basic operations of the Source class.

The Source class has many other methods that use one or more of the basic methods to perform operations such as selecting elements of the base Source by value or by position, or sorting elements. Many of the examples in this chapter and in Chapter 5, "Understanding Source Objects" use some of these methods. Other Source methods get objects that have information about the Source, such as the getDefinition, getInputs, and getType methods, or convert the values of the Source from one data type to another, such as the toDoubleSource method.

This section describes the basic Source methods and provides some examples of their use. Table 6–1 lists the basic Source methods.

Table 6–1 The Basic Source Methods

Method	Description
alias	Produces a Source that has the same elements as the base Source, but has the base Source as the type.

Table 6–1 (Cont.) The Basic Source Methods

Method	Description
distinct	Produces a Source that has the same elements as the base Source, except that any elements that are duplicated in the base appear only once in the derived Source.
join	Produces a Source that has the elements of the base Source that are specified by the joined, comparison, and comparisonRule parameters of the method call. If the visible parameter is true, then the joined Source is an output of the resulting Source.
position	Produces a Source that has the positions of the elements of the base Source, and that has the base Source as a regular input.
recursiveJoin	Similar to the join method, except that this method, in the Source that it produces, orders the elements of the Source hierarchically by parent-child relationships.
value	Produces a Source that has the same elements as the base Source, but that has the base Source as a regular input.

Using the Basic Methods

This section provides examples of using some of the basic methods.

Using the alias Method

You use the alias method to control the matching of a Source to an input. For example, if you want to find out if the measure values specified by an element of a dimension of the measure are greater than the measure values specified by the other elements of the same dimension, then you need to match the inputs of the measure twice in the same join operation. To do so, you can produce two Source objects that are aliases for the same dimension, make them inputs of two instances of the measure, join each measure instance to the associated aliased dimension, and then compare the results.

Example 6–1 performs such an operation. It produces a Source that specifies whether the number of units sold for each value of the Channel dimension is greater than the number of units sold for the other values of the Channel dimension.

The example joins to units, which is the Source for a measure, Source objects that are selections of single values of three of the dimensions of the measure to produce unitsSel. The unitsSel Source specifies the units elements for the dimension values that are specified by the timeSel, custSel, and prodSel objects, which are outputs of unitsSel.

The timeSel, custSel, and prodSel Source objects specify single values from the default hierarchies of the Time, Customer, and Product dimensions, respectively. The timeSel value is CALENDAR_YEAR::MONTH::2001.01, which identifies the month January, 2001, the custSel value is SHIPMENTS::SHIP_TO::BUSN WRLD SJ, which identifies the Business Word San Jose customer, and the prodSel value is PRODUCT_PRIMARY::ITEM::ENVY ABM, which identifies the Envoy Ambassador portable PC.

The example next creates two aliases, chanAlias1 and chanAlias2, for chanHier, which is the default hierarchy of the Channel dimension. It then produces unitsSel1 by joining unitsSel to the Source that results from calling the value method of chanAlias1. The unitsSel1 Source has the elements and outputs of unitsSel and it has chanAlias1 as an input. Similarly, the example produces unitsSel2, which has chanAlias2 as an input.

The example uses the gt method of unitsSel1, which determines whether the values of unitsSel1 are greater than the values of unitsSel2. The final join operations match chanAlias1 to the input of unitsSel1 and match chanAlias1 to the input of unitsSel2.

Example 6–1 Controlling Input-to-Source Matching With the alias Method

```
Source unitsSel = units.join(timeSel).join(custSel).join(prodSel);
Source chanAlias1 = chanHier.alias();
Source chanAlias2 = chanHier.alias();
NumberSource unitsSel1 = (NumberSource)
                          unitsSel.join(chanAlias1.value());
NumberSource unitsSel2 = (NumberSource)
                          unitsSel.join(chanAlias2.value());
Source result = unitsSel1.gt(unitsSel2)
                         .join(chanAlias1)   // Output 2, column
                         .join(chanAlias2);  // Output 1, row;
```

The result Source specifies the query, "Are the units sold values of unitsSel1 for the channel values of chanAlias1 greater than the units sold values of unitsSel2 for the channel values of chanAlias2?" Because result is produced by the joining of chanAlias2 to the Source produced by unitsSel1.gt(unitsSel2).join(chanAlias1), chanAlias2 is the first output of result, and chanAlias1 is the second output of result.

A Cursor for the result Source has as values the boolean values that answer the query. The values of the first output of the Cursor are the channel values specified by chanAlias2 and the values of the second output are the channel values specified by chanAlias1.

The following is a display of the values of the Cursor formatted as a crosstab with headings added. The column edge values are the values from chanAlias1, and the row edge values are the values from chanAlias2. The values of the crosstab cells are the boolean values that indicate whether the units sold value for the column channel value is greater than the units sold value for the row channel value. For example, the crosstab values in the first column indicate that the units sold for the column channel value Total Channel is not greater than the units sold for the row Total Channel value but it is greater than the units sold for the Direct Sales, Catalog, and Internet row values.

```
               ----------------- chanAlias1 ----------------
chanAlias2     TotalChannel  Catalog  Direct Sales    Internet
------------   ------------  -------  ------------  --------
TotalChannel      false       false      false       false
Catalog           true        false      false       false
Direct Sales      true        true       false       false
Internet          true        true       true        false
```

Using the distinct Method

You use the distinct method to produce a Source that does not have any duplicated values. Example 6–2 selects an element from a hierarchy of the Customer dimension and gets the descendants of that element. It then appends the descendants to the hierarchy element selection. Because the Source for the descendants includes the ancestor value, the example uses the distinct method to remove the duplicated ancestor value, which would otherwise appear twice in the result.

In Example 6–2, markets is a StringSource that represents the Markets hierarchy of the Customer dimension. The marketsAncestors object is the Source for the ancestors attribute of that hierarchy. To get a Source that represents the descendants of the ancestors, the example uses the join method to select, for each element of marketsAncestors, the elements of markets that have the marketsAncestors element as their ancestor. The join operation matches the base Source, markets, to the input of the ancestors attribute.

The resulting Source, marketsDescendants, however, still has markets as an input because the Source produced by the markets.value() method is the comparison Source of the join operation. The comparison parameter Source of a join operation does not participate in the matching of an input to a Source.

The selectValue method of markets selects the element of markets that has the value MARKETS::ACCOUNT::BUSN WRLD, which is the Business World account, and produces selVal. The join method of marketsDescendants uses selVal as the comparison Source. The method produces selValDescendants, which has the elements of marketsDescendants that are present in markets, and that are also in selVal. The input of marketsDescendants is matched by the joined Source markets. The markets Source is not an output of selValDescendants because the value of the visible parameter of the join operation is false.

The appendValues method of selVal produces selValPlusDescendants, which is the result of appending the elements of selValDescendants to the element of selVal and then removing any duplicate elements with the distinct method.

Example 6–2 Using the distinct Method

```
Source marketsDescendants =
                     markets.join(marketsAncestors, markets.value());
Source selVal = markets.selectValue("MARKETS::ACCOUNT::BUSN WRLD");
Source selValDescendants = marketsDescendants.join(markets,
                                                   selVal,
                                                   false);
Source selValPlusDescendants = selVal.appendValues(selValDescendants)
                                     .distinct();
```

A Cursor for the selValPlusDescendants Source has the following values:

```
MARKETS::ACCOUNT::BUSN WRLD
MARKETS::SHIP_TO::BUSN WRLD HAM
MARKETS::SHIP_TO::BUSN WRLD NAN
MARKETS::SHIP_TO::BUSN WRLD NY
MARKETS::SHIP_TO::BUSN WRLD SJ
```

If the example did not include the distinct method call, then a Cursor for selValPlusDescendants would have the following values:

```
MARKETS::ACCOUNT::BUSN WRLD
MARKETS::ACCOUNT::BUSN WRLD
MARKETS::SHIP_TO::BUSN WRLD HAM
MARKETS::SHIP_TO::BUSN WRLD NAN
MARKETS::SHIP_TO::BUSN WRLD NY
MARKETS::SHIP_TO::BUSN WRLD SJ
```

Using the join Method

You use the join method to produce a Source that has the elements of the base Source that are determined by the joined, comparison, and comparisonRule parameters of the method. The `visible` parameter determines whether the `joined` `Source` is an output of the `Source` produced by the join operation. You also use the `join` method to match a `Source` to an input of the base or `joined` parameter `Source`.

The `join` method has many signatures that are convenient shortcuts for the full `join(Source joined, Source comparison, int comparisonRule, boolean visible)` method. The examples in this chapter use various `join` method signatures.

The `Source` class has several constants that you can provide as the value of the `comparisonRule` parameter. Example 6–3 and Example 6–4 demonstrate the use of two of those constants, `COMPARISON_RULE_REMOVE` and `COMPARISON_RULE_DESCENDING`. Example 6–5 also uses `COMPARISON_RULE_REMOVE`.

Example 6–3 produces a result similar to Example 6–2. It uses `markets`, which is the `Source` for a hierarchy of the Customer dimension, and `marketsAncestors`, which is the `Source` for the ancestors attribute for the hierarchy. It also uses `marketsDescendants`, which is a `Source` for the descendants of elements of the hierarchy.

The example first selects an element of the hierarchy. Next, the `join` method of `marketsDescendants` produces `marketsDescendantsOnly`, which specifies the descendants of `markets`, and which has `markets` as an input because the `comparison` parameter of the join operation is the `Source` that results from the `markets.value()` method.

Because `COMPARISON_RULE_REMOVE` is the comparison rule of the join operation that produced `marketsDescendantsOnly`, a join operation that matches a `Source` to the input of `marketsDescendantsOnly` produces a `Source` that has only those elements of `marketsDescendantsOnly` that are not in the comparison `Source` of the join operation.

The next join operation performs such a match. It matches the joined `Source`, `markets`, to the input of `marketsDescendantsOnly` to produce `selValDescendantsOnly`, which specifies the descendants of the selected hierarchy value but does not include the selected value because `marketsDescendantsOnly` specifies the removal of any values that match the value of the comparison `Source`, which is `selVal`.

As a contrast, the last join operation produces `selValDescendants`, which specifies the descendants of the selected hierarchy value and which does include the selected value.

Example 6–3 Using COMPARISON_RULE_REMOVE

```
Source selVal = markets.selectValue("MARKETS::ACCOUNT::BUSN WRLD");
Source marketsDescendantsOnly =
      marketsDescendants.join(marketsDescendants.getDataType().value(),
                              markets.value(),
                              Source.COMPARISON_RULE_REMOVE);

// Select the descendants of the specified element.
Source selValDescendants = marketsDescendants.join(markets, selVal);

// Select only the descendants of the specified element.
Source selValDescendantsOnly = marketsDescendantsOnly.join(markets,
                                                           selVal);
```

A Cursor for selValDescendants has the following values.

```
MARKETS::ACCOUNT::BUSN WRLD
MARKETS::SHIP_TO::BUSN WRLD HAM
MARKETS::SHIP_TO::BUSN WRLD NAN
MARKETS::SHIP_TO::BUSN WRLD NY
MARKETS::SHIP_TO::BUSN WRLD SJ
```

A Cursor for selValDescendantsOnly has the following values.

```
MARKETS::SHIP_TO::BUSN WRLD HAM
MARKETS::SHIP_TO::BUSN WRLD NAN
MARKETS::SHIP_TO::BUSN WRLD NY
MARKETS::SHIP_TO::BUSN WRLD SJ
```

Example 6–4 demonstrates another join operation, which uses the comparison rule COMPARISON_RULE_DESCENDING. It uses the following Source objects.

- prodSelWithShortDescr, which is the Source produced by joining the Source for the short value description attribute of the Product dimension to the Source for the Family level of the Product Primary hierarchy of that dimension.
- unitPrice, which is the Source for the Unit Price measure.
- timeSelWithShortDescr, which is the Source produced by joining the Source for the short value description attribute of the Time dimension to the Source for a selected element of the Calendar Year hierarchy of that dimension.

The resulting Source specifies the product family level elements in descending order of total unit prices for the month of May, 2001.

Example 6–4 Using COMPARISON_RULE_DESCENDING

```
Source result =
       prodSelWithShortDescr.join(unitPrice,
                                  unitPrice.getDataType(),
                                  Source.COMPARISON_RULE_DESCENDING,
                                  true)
                            .join(timeSelWithShortDescr);
```

A Cursor for the result Source has the following values, displayed as a table. The table includes only the short value descriptions of the dimension elements and the unit price values, and has formatting added.

```
May, 2001

Total Unit Prices   Product Family
-----------------   --------------
     2,845.59       Portable PCs
     1,871.03       Desktop PCs
       415.37       Memory
       397.62       Monitors
       196.05       CD/DVD
       318.61       Modems/Fax
        74.68       Documentation
        65.92       Operating Systems
        36.50       Accessories
```

Using the position Method

You use the `position` method to produce a `Source` that has the positions of the elements of the base and has the base as an input. Example 6–5 uses the `position` method in producing a `Source` that specifies the selection of the first and last elements of the levels of a hierarchy of the Time dimension.

In the example, `mdmTimeDim` is the `MdmPrimaryDimension` for the Time dimension. The example gets the level attribute and the default hierarchy of the dimension. It then gets `Source` objects for the attribute and the hierarchy.

Next, the example creates an array of `Source` objects and gets a `List` of the `MdmHierarchyLevel` components of the hierarchy. It gets the `Source` object for each level and adds it to the array, and then creates a list `Source` that has the `Source` objects for the levels as element values.

The example then produces `levelMembers`, which is a `Source` that specifies the members of the levels of the hierarchy. Because the `comparison` parameter of the join operation is the `Source` produced by `levelList.value()`, `levelMembers` has `levelList` as an input. Therefore, `levelMembers` is a `Source` that returns the members of each level, by level, when the input is matched in a join operation.

The `range Source` specifies a range of elements from the second element to the next to last element of a `Source`.

The next join operation produces the `firstAndLast Source`. The base of the operation is `levelMembers`. The `joined` parameter is the `Source` that results from the `levelMembers.position()` method. The `comparison` parameter is the `range Source` and the comparison rule is COMPARISON_RULE_REMOVE. The value of the `visible` parameter is `true`. The `firstAndLast Source` therefore specifies only the first and last members of the levels because it removes all of the other members of the levels from the selection. The `firstAndLast Source` still has `levelList` as an input.

The final join operation matches the input of `firstAndLast` to `levelList`.

Example 6–5 Selecting the First and Last Time Elements

```
MdmAttribute mdmTimeLevelAttr = mdmTimeDim.getLevelAttribute();
MdmLevelHierarchy mdmTimeHier = (MdmLevelHierarchy)
                                 mdmTimeDim.getDefaultHierarchy();

Source levelRel = mdmTimeLevelAttr.getSource();
StringSource calendar = (StringSource) mdmTimeHier.getSource();

Source[] levelSources = new Source[3];
List levels = mdmTimeHier.getHierarchyLevels();
for (int i = 0; i < levelSources.length; i++)
{
  levelSources[i] = ((MdmHierarchyLevel) levels.get(i)).getSource();
}
Source levelList = dp.createListSource(levelSources);
Source levelMembers =  calendar.join(levelRel, levelList.value());
Source range = dp.createRangeSource(2, levelMembers.count().minus(1));
Source firstAndLast = levelMembers.join(levelMembers.position(),
                                        range
                                        Source.COMPARISON_RULE_REMOVE,
                                        true);

Source result = firstAndLast.join(levelList);
```

A Cursor for the result Source has the following values, displayed as a table with column headings and formatting added. The left column names the level, the middle column is the position of the member in the level, and the right column is the local value of the member.

```
Level       Member Position in Level  Member Value
----------  ------------------------  ------------
TOTAL_TIME              1               TOTAL
YEAR                    1               CY1998
YEAR                   10               CY2007
QUARTER                 1               CY1998.Q1
QUARTER                40               CY2007.Q4
MONTH                   1               1998.01
MONTH                 120               2007.12
```

Using the recursiveJoin Method

You use the recursiveJoin method to produce a Source that has elements that are ordered hierarchically. You use the recursiveJoin method only with the Source for an MdmHierarchy or with a subtype of such a Source. The method produces a Source whose elements are ordered hierarchically by the parents and their children in the hierarchy.

Like the join method, you use the recursiveJoin method to produce a Source that has the elements of the base Source that are determined by the joined, comparison, and comparisonRule parameters of the method. The visible parameter determines whether the joined Source is an output of the Source produced by the recursive join operation.

The recursiveJoin method has several signatures. The full recursiveJoin method has parameters that specify the parent attribute of the hierarchy, whether the result should have the parents before or after their children, how to order the elements of the result if the result includes children but not the parent, and whether the joined Source is an output of the resulting Source.

Example 6–6 uses a recursiveJoin method that lists the parents first, restricts the parents to the base, and does not add the joined Source as an output. The example first sorts the elements of a hierarchy of the Product dimension by hierarchical levels and then by the value of the package attribute of each element.

The first recursiveJoin method orders the elements of the prodHier hierarchy in ascending hierarchical order. The prodParent object is the Source for the parent attribute of the hierarchy.

The prodPkgAttr object in the second recursiveJoin method is the Source for the package attribute of the dimension. Only the elements of the Item level have a related package value. Because the elements in the aggregate levels Total Product, Class, and Family, do not have a related package, the package attribute value for elements in those levels is null, which appears as NA in the results. Some of the Item level elements do not have a related package, so their values are NA, also.

The second recursiveJoin method joins the package attribute values to their related hierarchy elements and sorts the elements hierarchically by level, and then sorts them in ascending order in the level by the package value. The COMPARISON_RULE_ASCENDING_NULLS_FIRST parameter specifies that elements that have a null value appear before the other elements in the same level. The example then joins the result of the method, sortedHierNullsFirst, to the package attribute to produce a Source that has the package values as element values and sortedHierNullsFirst as an output.

The third `recursiveJoin` method is the same as the second, except that the `COMPARISON_RULE_ASCENDING_NULLS_LAST` parameter specifies that elements that have a `null` value appear after the other elements in the same level.

Example 6–6 Sorting Products Hierarchically By Attribute

```
Source result1 = prodHier.recursiveJoin(prodDim.value(),
                                        prodHier.getDataType(),
                                        prodParent,
                                        Source.COMPARISON_RULE_ASCENDING);

Source sortedHierNullsFirst =
      prodHier.recursiveJoin(prodPkgAttr,
                             prodPkgAttr.getDataType(),
                             prodParent,
                             Source.COMPARISON_RULE_ASCENDING_NULLS_FIRST);
Source result2 = prodPkgAttr.join(sortedHierNullsFirst);

Source sortedHierNullsLast =
      prodHier.recursiveJoin(prodPkgAttr,
                             prodPkgAttr.getDataType(),
                             prodParent,
                             Source.COMPARISON_RULE_DESCENDING_NULLS_LAST);
Source result3 = prodPkgAttr.join(sortedHierNullsLast);
```

A `Cursor` for the `result1 Source` has the following values, displayed with a heading added. The list contains only the first seventeen values of the `Cursor`.

```
Product Dimension Element Value
--------------------------------
PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL
PRODUCT_PRIMARY::CLASS::HRD
PRODUCT_PRIMARY::FAMILY::DISK
PRODUCT_PRIMARY::ITEM::EXT CD ROM
PRODUCT_PRIMARY::ITEM::EXT DVD
PRODUCT_PRIMARY::ITEM::INT 8X DVD
PRODUCT_PRIMARY::ITEM::INT CD ROM
PRODUCT_PRIMARY::ITEM::INT CD USB
PRODUCT_PRIMARY::ITEM::INT RW DVD
PRODUCT_PRIMARY::FAMILY::DTPC
PRODUCT_PRIMARY::ITEM::SENT FIN
PRODUCT_PRIMARY::ITEM::SENT MM
PRODUCT_PRIMARY::ITEM::SENT STD
PRODUCT_PRIMARY::FAMILY::LTPC
PRODUCT_PRIMARY::ITEM::ENVY ABM
PRODUCT_PRIMARY::ITEM::ENVY EXE
PRODUCT_PRIMARY::ITEM::ENVY STD
...
```

A `Cursor` for the `result2 Source` has the following values, displayed as a table with headings added. The table contains only the first seventeen values of the `Cursor`. The left column has the member values of the hierarchy and the right column has the package attribute value for the member.

The Item level elements that have a `null` value appear first, and then the other level members appear in ascending order of package value. Since the data type of the package attribute is String, the package values are in ascending alphabetical order.

```
Product Dimension Element Value        Package Attribute Value
-------------------------------------  -----------------------
PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL     NA
PRODUCT_PRIMARY::CLASS::HRD               NA
PRODUCT_PRIMARY::FAMILY::DISK             NA
PRODUCT_PRIMARY::ITEM::EXT CD ROM         NA
PRODUCT_PRIMARY::ITEM::INT 8X DVD         NA
PRODUCT_PRIMARY::ITEM::INT CD USB         NA
PRODUCT_PRIMARY::ITEM::EXT DVD            Executive
PRODUCT_PRIMARY::ITEM::INT CD ROM         Laptop Value Pack
PRODUCT_PRIMARY::ITEM::INT RW DVD         Multimedia
PRODUCT_PRIMARY::FAMILY::DTPC             NA
PRODUCT_PRIMARY::ITEM::SENT FIN           NA
PRODUCT_PRIMARY::ITEM::SENT STD           NA
PRODUCT_PRIMARY::ITEM::SENT MM            Multimedia
PRODUCT_PRIMARY::FAMILY::LTPC             NA
RODUCT_PRIMARY::ITEM::ENVY ABM            NA
PRODUCT_PRIMARY::ITEM::ENVY EXE           Executive
PRODUCT_PRIMARY::ITEM::ENVY STD           Laptop Value Pack
...
```

A `Cursor` for the `result3 Source` has the following values, displayed as a table with headings added. This time the members are in descending order, alphabetically by package attribute value.

```
Product Dimension Element Value        Package Attribute Value
-------------------------------------  -----------------------
PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL     NA
PRODUCT_PRIMARY::CLASS::HRD               NA
PRODUCT_PRIMARY::FAMILY::DISK             NA
PRODUCT_PRIMARY::ITEM::EXT CD ROM         NA
PRODUCT_PRIMARY::ITEM::INT 8X DVD         NA
PRODUCT_PRIMARY::ITEM::INT CD USB         NA
PRODUCT_PRIMARY::ITEM::INT RW DVD         Multimedia
PRODUCT_PRIMARY::ITEM::INT CD ROM         Laptop Value Pack
PRODUCT_PRIMARY::ITEM::EXT DVD            Executive
PRODUCT_PRIMARY::FAMILY::DTPC             NA
PRODUCT_PRIMARY::ITEM::SENT FIN           NA
PRODUCT_PRIMARY::ITEM::SENT STD           NA
PRODUCT_PRIMARY::ITEM::SENT MM            Multimedia
PRODUCT_PRIMARY::FAMILY::LTPC             NA
PRODUCT_PRIMARY::ITEM::ENVY ABM           NA
PRODUCT_PRIMARY::ITEM::ENVY STD           Laptop Value Pack
PRODUCT_PRIMARY::ITEM::ENVY EXE           Executive
...
```

Using the value Method

You use the `value` method to create a `Source` that has itself as an input. That relationship enables you to select a subset of elements of the `Source`.

Example 6–7 demonstrates the selection of such a subset. In the example, `shipHier` is a `Source` for the `SHIPMENTS` hierarchy of the Customer dimension. The `selectValues` method of `shipHier` produces `custSel`, which is a selection of some of the elements of `shipHier`. The `selectValues` method of `custSel` produces `custSel2`, which is a subset of that selection.

The first `join` method has `custSel` as the base and as the joined `Source`. It has `custSel2` as the comparison `Source`. The elements of the resulting `Source`, `result1`, are one set of the elements of `custSel` for each element of `custSel` that is in the comparison `Source`. The `true` value of the `visible` parameter causes the joined `Source` to be an output of `result1`.

The second `join` method also has `custSel` as the base and `custSel2` as the comparison `Source`, but it has the result of the `custSel.value()` method as the joined `Source`. Because `custSel` is an input of the joined `Source`, the base `Source` matches that input. That input relationship causes the resulting `Source`, `result2`, to have only those elements of `custSel` that are also in the comparison `Source`.

Example 6–7 Selecting a Subset of the Elements of a Source

```
StringSource custSel = (StringSource) shipHier.selectValues(new String[]
                                         {"SHIPMENTS::SHIP_TO::COMP WHSE SIN",
                                          "SHIPMENTS::SHIP_TO::COMP WHSE LON",
                                          "SHIPMENTS::SHIP_TO::COMP WHSE SJ",
                                          "SHIPMENTS::SHIP_TO::COMP WHSE ATL"});

Source custSel2 = custSel.selectValues(new String[]
                                         {"SHIPMENTS::SHIP_TO::COMP WHSE SIN",
                                          "SHIPMENTS::SHIP_TO::COMP WHSE SJ"});

Source result1 = custSel.join(custSel, custSel2, true);

Source  result2 = custSel.join(custSel.value(), custSel2, true);
```

A `Cursor` for `result1` has the following values, displayed as a table with headings added. The left column has the values of the elements of the output of the `Cursor`. The right column has the values of the `Cursor`.

```
          Output Value                           result1 Value
---------------------------------   ---------------------------------
SHIPMENTS::SHIP_TO::COMP WHSE SJ    SHIPMENTS::SHIP_TO::COMP WHSE ATL
SHIPMENTS::SHIP_TO::COMP WHSE SJ    SHIPMENTS::SHIP_TO::COMP WHSE SJ
SHIPMENTS::SHIP_TO::COMP WHSE SJ    SHIPMENTS::SHIP_TO::COMP WHSE SIN
SHIPMENTS::SHIP_TO::COMP WHSE SJ    SHIPMENTS::SHIP_TO::COMP WHSE LON
SHIPMENTS::SHIP_TO::COMP WHSE SIN   SHIPMENTS::SHIP_TO::COMP WHSE ATL
SHIPMENTS::SHIP_TO::COMP WHSE SIN   SHIPMENTS::SHIP_TO::COMP WHSE SJ
SHIPMENTS::SHIP_TO::COMP WHSE SIN   SHIPMENTS::SHIP_TO::COMP WHSE SIN
SHIPMENTS::SHIP_TO::COMP WHSE SIN   SHIPMENTS::SHIP_TO::COMP WHSE LON
```

A `Cursor` for `result2` has the following values, displayed as a table with headings added. The left column has the values of the elements of the output of the `Cursor`. The right column has the values of the `Cursor`.

```
          Output Value                           result2 Value
---------------------------------   ---------------------------------
SHIPMENTS::SHIP_TO::COMP WHSE SJ    SHIPMENTS::SHIP_TO::COMP WHSE SJ
SHIPMENTS::SHIP_TO::COMP WHSE SIN   SHIPMENTS::SHIP_TO::COMP WHSE SIN
```

Using Other Source Methods

Along with the methods that are various signatures of the basic methods, the `Source` class has many other methods that use combinations of the basic methods. Some methods perform selections based on a single position, such as the `at` and `offset` methods. Others operate on a range of positions, such as the `interval` method. Some perform comparisons, such as `eq` and `gt`, select one or more elements, such as `selectValue` or `removeValue`, or sort elements, such as `sortAscending` or `sortDescendingHierarchically`.

The subclasses of Source each have other specialized methods, also. For example, the NumberSource class has many methods that perform mathematical functions such as abs, div, and cos, and methods that perform aggregations, such as average and total.

This section has examples that demonstrate the use of some of the Source methods. Some of the examples are tasks that an OLAP application typically performs.

Using the extract Method

You use the extract method to extract the values of a Source that has Source objects as element values. If the elements of a Source have element values that are not Source objects, then the extract method operates like the value method.

Example 6–8 uses the extract method to get the values of the NumberSource objects that are themselves the values of the elements of measDim. Each of the NumberSource objects represents a measure.

The example selects values from hierarchies of the dimensions of the NumberSource for the Units and Sales measures. Two of those dimensions are the dimensions of the NumberSource for the Unit Price measure.

Next, the example creates a list Source, measDim, which has the three NumberSource objects as the element values. It then uses the extract method to get the values of the NumberSource objects. The resulting unnamed Source has measDim as an extraction input. The input is matched by first join operation, which has measDim as the joined parameter. The example then matches the other inputs of the measures by joining the dimension selections to produce the result Source.

Example 6–8   Using the extract Method

```
Source prodSel = prodHier.selectValues(new String[]
                                        {"PRODUCT_PRIMARY::ITEM::ENVY STD",
                                         "PRODUCT_PRIMARY::ITEM::ENVY EXE",
                                         "PRODUCT_PRIMARY::ITEM::ENVY ABM"});
Source chanSel = chanHier.selectValue("CHANNEL_PRIMARY::CHANNEL::DIR");
Source timeSel = timeHier.selectValue("CALENDAR_YEAR::MONTH::2001.05");
Source custSel = custHier.selectValue("SHIPMENTS::TOTAL_CUSTOMER::TOTAL");

Source measDim = dp.createListSource(new Source[] {units, unitPrice, sales});

Source result = measDim.extract().join(measDim)   // column
                                 .join(prodSel)   // row
                                 .join(timeSel)   // page
                                 .join(chanSel)   // page
                                 .join(custSel);  // page
```

The following crosstab displays the values of a Cursor for the result Source, with headings and formatting added.

```
SHIPMENTS::TOTAL_CUSTOMER::TOTAL
CHANNEL_PRIMARY::CHANNEL::DIR
CALENDAR_YEAR::MONTH::2001.05

ITEM         UNIT PRICE   UNITS SOLD   SALES AMOUNT
--------     ----------   ----------   -------------
ENVY ABM       2,993.29       26          77,825.54
ENVY EXE       3,147.85       37         116,470.45
ENVY STD       2,395.63       39          93,429.57
```

Creating a Cube and Pivoting Edges

One typical OLAP operation is the creation of a cube, which is a multi-dimensional array of data. The data of the cube is specified by the elements of the column, row, and page edges of the cube. The data of the cube can be data from a measure that is specified by the elements of the dimensions of the measure. The cube data can also be dimension elements that are specified by some calculation of the measure data, such as products that have unit sales quantities greater than a specified amount.

Most of the examples in this section create cubes. Example 6–9 creates a cube that has the quantity of units sold as the data of the cube. The column edge values are initially from a channel dimension hierarchy, the row edge values are from a time dimension hierarchy, and the page edge values of the cube are from elements of hierarchies for product and customer dimensions. The product and customer elements on the page edge are represented by parameterized `Source` objects.

The example joins the selections of the dimension elements to the short value description attributes for the dimensions so that the results have more information than just the numerical identifications of the dimension values. It then joins the `Source` objects derived from the dimensions to the `Source` for the measure to produce the `cube` query. It commits the current `Transaction`, and then creates a `Cursor` for the query and displays the values.

After displaying the values of the `Cursor`, the example changes the value of the `Parameter` for the parameterized `Source` for the customer selection, thereby retrieving a different result set using the same `Cursor` in the same `Transaction`. The example resets the position of the `Cursor`, and displays the values of the `Cursor` again.

The example then pivots the column and row edges so that the column values are time elements and the row values are channel elements. It commits the `Transaction`, creates another `Cursor` for the query, and displays the values. It then changes the value of each `Parameter` object and displays the values of the `Cursor` again.

The `dp` object is the `DataProvider`. The `getContext` method gets a `Context11g` object that has a method that displays the values of the `Cursor` in a crosstab format.

Example 6–9 Creating a Cube and Pivoting the Edges

```
// Create Parameter objects with values from the default hierarchies
// of the Customer and Product dimensions.
StringParameter custParam =
                      new StringParameter(dp, "SHIPMENTS::REGION::EMEA");
StringParameter prodParam =
                    new StringParameter(dp, "PRODUCT_PRIMARY::FAMILY::LTPC");

// Create parameterized Source objects using the Parameter objects.
Source custParamSrc = custParam.createSource();
Source prodParamSrc = prodParam.createSource();

// Select single values from the hierarchies, using the Parameter
// objects as the comparisons in the join operations.
Source paramCustSel = custHier.join(custHier.value(), custParamSrc);
Source paramProdSel = prodHier.join(prodHier.value(), prodParamSrc);

// Select elements from the other dimensions of the measure
Source timeSel = timeHier.selectValues(new String[]
                                              {"CALENDAR_YEAR::YEAR::CY1999"
                                               "CALENDAR_YEAR::YEAR::CY2000",
                                               "CALENDAR_YEAR::YEAR::CY2001"});
```

```
Source chanSel = chanHier.selectValues(new String[]
                                            {"CHANNEL_PRIMARY::CHANNEL::DIR",
                                             "CHANNEL_PRIMARY::CHANNEL::CAT",
                                             "CHANNEL_PRIMARY::CHANNEL::INT"});

// Join the dimension selections to the short description attributes
// for the dimensions.
Source columnEdge = chanSel.join(chanShortDescr);
Source rowEdge = timeSel.join(timeShortDescr);
Source page1 = paramProdSel.join(prodShortDescr);
Source page2 = paramCustSel.join(custShortDescr);

// Join the dimension selections to the measure.
Source cube = units.join(columnEdge)
                   .join(rowEdge)
                   .join(page2)
                   .join(page1);

// The following method commits the current Transaction.
getContext().commit();

// Create a Cursor for the query.
CursorManager cursorMngr = dp.createCursorManager(cube);
CompoundCursor cubeCursor = (CompoundCursor) cursorMngr.createCursor();

// Display the values of the Cursor as a crosstab.
getContext().displayCursorAsCrosstab(cubeCursor);

// Change the customer parameter value.
custParam.setValue("SHIPMENTS::REGION::AMER");

// Reset the Cursor position to 1 and display the values again.
cubeCursor.setPosition(1);
println();
getContext().displayCursorAsCrosstab(cubeCursor);

// Pivot the column and row edges.
columnEdge = timeSel.join(timeShortDescr);
rowEdge = chanSel.join(chanShortDescr);

// Join the dimension selections to the measure.
cube = units.join(columnEdge)
            .join(rowEdge))
            .join(page2)
            .join(page1);

// Commit the current Transaction.
getContext().commit();

// Create another Cursor.
cursorMngr = dp.createCursorManager(cube);
cubeCursor = (CompoundCursor) cursorMngr.createCursor();
getContext().displayCursorAsCrosstab(cubeCursor);

// Change the product parameter value.
prodParam.setValue("PRODUCT_PRIMARY::FAMILY::DTPC");

// Reset the Cursor position to 1
cubeCursor.setPosition(1);
println();
```

```
getContext().displayCursorAsCrosstab(cubeCursor);
```

The following crosstab has the values of `cubeCursor` displayed by the first `displayCursorAsCrosstab` method.

```
Portable PCs
Europe

        Catalog  Direct Sales  Internet
1999       1986            86         0
2000       1777           193        10
2001       1449           196       215
```

The following crosstab has the values of `cubeCursor` after the example changed the value of the `custParam Parameter` object.

```
Portable PCs
North America

        Catalog  Direct Sales  Internet
1999       6841           385         0
2000       6457           622        35
2001       5472           696       846
```

The next crosstab has the values of `cubeCursor` after pivoting the column and row edges.

```
Portable PCs
North America

                1999    2000    2001
Catalog         6841    6457    5472
Direct Sales     385     622     696
Internet           0      35     846
```

The last crosstab has the values of `cubeCursor` after changing the value of the `prodParam Parameter` object.

```
Desktop PCs
North America

                1999    2000    2001
Catalog        14057   13210   11337
Direct Sales     793    1224    1319
Internet           0      69    1748
```

Drilling Up and Down in a Hierarchy

Drilling up or down in a dimension hierarchy is another typical OLAP operation. Example 6–10 demonstrates getting the elements of one level of a dimension hierarchy, selecting an element, and then getting the parent, children, and ancestors of the element.

The example uses the following objects.

- `levelSrc`, which is the `Source` for the Family level of the Product Primary hierarchy of the Product dimension.
- `prodHier`, which is the `Source` for the Product Primary hierarchy.
- `prodHierParentAttr`, which is the `Source` for the parent attribute of the hierarchy.

- `prodHierAncsAttr`, which is the `Source` for the ancestors attribute of the hierarchy.
- `prodShortLabel`, which is the `Source` for the short value description attribute of the Product dimension.

Example 6–10 Drilling in a Hierarchy

```
int pos = 5;
// Get the element at the specified position of the level Source.
Source levelElement = levelSrc.at(pos);

// Select the element of the hierarchy with the specified value.
Source levelSel = prodHier.join(prodHier.value(), levelElement);

// Get ancestors of the level element.
Source levelElementAncs = prodHierAncsAttr.join(prodHier, levelElement);
// Get the parent of the level element.
Source levelElementParent = prodHierParentAttr.join(prodHier, levelElement);
// Get the children of a parent.
Source prodHierChildren = prodHier.join(prodHierParentAttr, prodHier.value());

// Select the children of the level element.
Source levelElementChildren = prodHierChildren.join(prodHier, levelElement);


// Get the short value descriptions for the elements of the level.
Source levelSrcWithShortDescr = prodShortLabel.join(levelSrc);

// Get the short value descriptions for the children.
Source levelElementChildrenWithShortDescr =
                               prodShortLabel.join(levelElementChildren);

// Get the short value descriptions for the parents.
Source levelElementParentWithShortDescr =
               prodShortLabel.join(prodHier, levelElementParent, true);

// Get the short value descriptions for the ancestors.
Source levelElementAncsWithShortDescr =
              prodShortLabel.join(prodHier, levelElementAncs, true);

// Commit the current Transaction.
getContext().commit();

// Create Cursor objects and display their values.
println("Level element values:");
getContext().displayResult(levelSrcWithShortDescr);
println("\nLevel element at position " + pos + ":");
getContext().displayResult(levelElement);
println("\nParent of the level element:");
getContext().displayResult(levelElementParent);
println("\nChildren of the level element:");
getContext().displayResult(levelElementChildrenWithShortDescr);
println("\nAncestors of the level element:");
getContext().displayResult(levelElementAncs);
```

The following list has the values of the `Cursor` objects created by the `displayResults` methods.

```
Level element values:
```

```
1: (PRODUCT_PRIMARY::FAMILY::ACC,Accessories)
2: (PRODUCT_PRIMARY::FAMILY::DISK,CD/DVD)
3: (PRODUCT_PRIMARY::FAMILY::DOC,Documentation)
4: (PRODUCT_PRIMARY::FAMILY::DTPC,Portable PCs)
5: (PRODUCT_PRIMARY::FAMILY::LTPC,Desktop PCs)
6: (PRODUCT_PRIMARY::FAMILY::MEM,Memory)
7: (PRODUCT_PRIMARY::FAMILY::MOD,Modems/Fax)
8: (PRODUCT_PRIMARY::FAMILY::MON,Monitors)
9: (PRODUCT_PRIMARY::FAMILY::OS,Operating Systems)

Level element at position 5:

1: PRODUCT_PRIMARY::FAMILY:LTPC

Parent of the level element:

1: (PRODUCT_PRIMARY::CLASS::HRD,Hardware)

Children of the level element:

1: (PRODUCT_PRIMARY::ITEM::ENVY ABM,Envoy Ambassador)
2: (PRODUCT_PRIMARY::ITEM::ENVY EXE,Envoy Executive)
3: (PRODUCT_PRIMARY::ITEM::ENVY STD,Envoy Standard)

Ancestors of the level element:

1: (PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL,Total Product)
2: (PRODUCT_PRIMARY::CLASS::HRD,Hardware)
3: (PRODUCT_PRIMARY::FAMILY::LTPC,Portable PCs)
```

Sorting Hierarchically by Measure Values

Example 6–11 uses the recursiveJoin method to sort the elements of the Product Primary hierarchy of the Product dimension hierarchically in ascending order of the values of the Units measure. The example joins the sorted products to the short value description attribute of the dimension, and then joins the result of that operation, sortedProductsShortDescr, to units.

The successive joinHidden methods join the selections of the other dimensions of units to produce the result Source, which has the measure data as element values and sortedProductsShortDescr as an output. The example uses the joinHidden methods so that the other dimension selections are not outputs of the result.

The example uses the following objects.

- prodHier, which is the Source for the Product Primary hierarchy.
- units, which is the Source for the Units measure of product units sold.
- prodParent, which is the Source for the parent attribute of the Product Primary hierarchy.
- prodShortDescr, which is the Source for the short value description attribute of the Product dimension.
- custSel, which is a Source that specifies a single element of the default hierarchy of the Customer dimension. The value of the element is SHIPMENTS::TOTAL_CUSTOMER::TOTAL, which is the total for all customers.
- chanSel, which is a Source that specifies a single element of the default hierarchy of the Channel dimension. The value of the element value is CHANNEL_PRIMARY::CHANNEL::DIR, which is the direct sales channel.

- `timeSel`, which is a `Source` that specifies a single element of the default hierarchy of the Time dimension. The value of the element value is `CALENDAR_YEAR::YEAR::CY2001`, which is the year 2001.

Example 6–11 Hierarchical Sorting by Measure Value

```
Source sortedProduct =
            prodHier.recursiveJoin(units,
                                   units.getDataType(),
                                   prodParent,
                                   Source.COMPARISON_RULE_ASCENDING,
                                   true,  // Parents first
                                   true); // Restrict parents to base

Source sortedProductShortDescr = prodShortDescr.join(sortedProduct);
Source result = units.join(sortedProductShortDescr)
                     .joinHidden(custSel)
                     .joinHidden(chanSel)
                     .joinHidden(timeSel);
```

A `Cursor` for the `result Source` has the following values, displayed in a table with column headings and formatting added. The left column has the name of the level in the `PRODUCT_PRIMARY` hierarchy. The next column to the right has the product identification value, and the next column has the short value description of the product. The rightmost column has the number of units of the product sold to all customers in the year 2001 through the direct sales channel.

The table contains only the first nine and the last eleven values of the `Cursor`, plus the Software/Other class value. The product values are listed in hierarchical order by units sold. The Hardware class appears before the Software/Other class because the Software/Other class has a greater number of units sold. In the Hardware class, the Portable PCs family sold the fewest units, so it appears first. In the Software/Other class, the Accessories family has the greatest number of units sold, so it appears last.

```
Product Level      ID              Description           Units Sold
-------------  ------------   ----------------------------- ----------
TOTAL_PRODUCT  TOTAL          Total Product                    43,785
CLASS          HRD            Hardware                         16,543
FAMILY         LTPC           Portable PCs                      1,192
ITEM           ENVY ABM       Envoy Ambassador                    330
ITEM           ENVY EXE       Envoy Executive                     385
ITEM           ENVY STD       Envoy Standard                      477
FAMILY         MON            Monitors                          1,193
ITEM           19 SVGA        Monitor- 19" Super VGA              207
ITEM           17 SVGA        Monitor- 17"Super VGA               986
...
CLASS          SFT            Software/Other)                  27,242
...
FAMILY         ACC            Accessories                      18,949
ITEM           ENVY EXT KBD   Envoy External Keyboard             146
ITEM           EXT KBD        External 101-key keyboard           678
ITEM           MM SPKR 5      Multimedia speakers- 5" cones       717
ITEM           STD MOUSE      Standard Mouse                      868
ITEM           MM SPKR 3      Multimedia speakers- 3" cones     1,120
ITEM           144MB DISK     1.44MB External 3.5" Diskette     1,145
TEM            KBRD REST      Keyboard Wrist Rest               2,231
ITEM           LT CASE        Laptop carrying case              3,704
ITEM           DLX MOUSE      Deluxe Mouse                      3,884
ITEM           MOUSE PAD      Mouse Pad                         4,456
```

Using NumberSource Methods To Compute the Share of Units Sold

Example 6–12 uses the NumberSource methods div and times to produce a Source that specifies the share that the Desktop PC and Portable PC families have of the total quantity of product units sold for the selected time, customer, and channel values. The example first uses the selectValue method of prodHier, which is the Source for a hierarchy of the Product dimension, to produce totalProds, which specifies a single element with the value PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL, which is the highest aggregate level of the hierarchy.

The joinHidden method of the NumberSource units produces totalUnits, which specifies the Units measure values at the total product level, without having totalProds appear as an output of totalUnits. The div method of units then produces a Source that represents each units sold value divided by total quantity of units sold. The times method then multiplies the result of that div operation by 100 to produce productShare, which represents the percentage, or share, that a product element has of the total quantity of units sold. The productShare Source has the inputs of the units measure as inputs.

The prodFamilies object is the Source for the Family level of the Product Primary hierarchy. The join method of productShare, with prodFamilies as the joined Source, produces a Source that specifies the share that each product family has of the total quantity of products sold.

The custSel, chanSel, and timeSel Source objects are selections of single elements of hierarchies of the Customer, Channel, and Time dimensions. The remaining join methods match those Source objects to the other inputs of productShare, to produce result. The join(Source joined, String comparison) signature of the join method produces a Source that does not have the joined Source as an output.

The result Source specifies the share for each product family of the total quantity of products sold to all customers through the direct sales channel in the year 2001.

Example 6–12 Getting the Share of Units Sold

```
Source totalProds =
        prodHier.selectValue("PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL");
NumberSource totalUnits = (NumberSource) units.joinHidden(totalProds);
Source productShare = units.div(totalUnits).times(100);
Source result =
      productShare.join(prodFamilies)
                  .join(timeHier, "CALENDAR_YEAR::YEAR::CY2001")
                  .join(chanHier, "CHANNEL_PRIMARY::CHANNEL::DIR")
                  .join(custHier, "SHIPMENTS::TOTAL_CUSTOMER::TOTAL");
Source sortedResult = result.sortAscending();
```

A Cursor for the sortedResult Source has the following values, displayed in a table with column headings and formatting added. The left column has the product family value and the right column has the share of the total number of units sold for the product family to all customers through the direct sales channel in the year 2001.

```
Product Family Element        Share of Total Units Sold
-----------------------------  -------------------------
PRODUCT_PRIMARY::FAMILY::LTPC            2.72%
PRODUCT_PRIMARY::FAMILY::MON             2.73%
PRODUCT_PRIMARY::FAMILY::MEM             3.57%
PRODUCT_PRIMARY::FAMILY::DTPC            5.13%
PRODUCT_PRIMARY::FAMILY::DOC             6.4%
PRODUCT_PRIMARY::FAMILY::DISK           11.71%
```

```
PRODUCT_PRIMARY::FAMILY::MOD           11.92%
PRODUCT_PRIMARY::FAMILY::OS            12.54%
PRODUCT_PRIMARY::FAMILY::ACC           43.28%
```

Selecting Based on Time Series Operations

This section has two examples of using methods that operate on a series of time dimension elements. Example 6–13 uses the `lag` method of `unitPrice`, which is the `Source` for the Unit Price measure, to produce `unitPriceLag4`, which specifies, for each element of `unitPrice`, the element of `unitPrice` that is four time periods before it at the same time hierarchy level.

In the example, `dp` is the `DataProvider`. The `createListSource` method creates `measuresDim`, which has the `unitPrice` and `unitPriceLag4` `Source` objects as element values. The `extract` method of `measuresDim` gets the values of the elements of `measuresDim`. The `Source` produced by the `extract` method has `measuresDim` as an extraction input. The first `join` method matches a `Source`, `measuresDim`, to the input of the `Source` produced by the `extract` method.

The `unitPrice` and `unitPriceLag4` measures both have the Product and Time dimensions as inputs. The second `join` method matches `quarterLevel`, which is a `Source` for the Quarter level of the Calendar Year hierarchy of the Time dimension, to the measure input for the Time dimension, and makes it an output of the resulting `Source`.

The `joinHidden` method matches `prodSel` to the measure input for the Product dimension, and does not make `prodSel` an output of the resulting `Source`. The `prodSel` `Source` specifies the single hierarchy element `PRODUCT_PRIMARY::FAMILY::DTPC`, which is Desktop PCs.

The `lagResult` `Source` specifies the aggregate unit prices for each quarter and the aggregate unit prices for the quarter four quarters earlier for the Desktop PC product family.

Example 6–13 Using the Lag Method

```
NumberSource unitPriceLag4 = unitPrice.lag(mdmTimeHier, 4);
Source measuresDim = dp.createListSource(new Source[] {unitPrice,
                                                      unitPriceLag4});

Source lagResult = measuresDim.extract()
                              .join(measuresDim)
                              .join(quarterLevel)
                              .joinHidden(prodSel);
```

A `Cursor` for the `lagResult` `Source` has the following values, displayed in a table with column headings and formatting added. The left column has the quarter, the middle column has the total of the unit prices for the members of the Desktop PC family for that quarter, and the right column has the total of the unit prices for the quarter four quarters earlier. The first four values in the right column are `NA` because quarter 5, Q1-98, is the first quarter in the Calendar Year hierarchy. The table includes only the first eight quarters.

```
                                                Unit Price
Quarter                            Unit Price  Four Quarters Before
---------------------------------  ----------  --------------------
CALENDAR_YEAR::QUARTER::CY1998.Q1    2687.54               NA
CALENDAR_YEAR::QUARTER::CY1998.Q2    2704.48               NA
CALENDAR_YEAR::QUARTER::CY1998.Q3    2673.27               NA
CALENDAR_YEAR::QUARTER::CY1998.Q4    2587.76               NA
```

```
CALENDAR_YEAR::QUARTER::CY1999.Q1    2394.79          2687.54
CALENDAR_YEAR::QUARTER::CY1999.Q2    2337.18          2704.48
CALENDAR_YEAR::QUARTER::CY1999.Q3    2348.39          2673.27
CALENDAR_YEAR::QUARTER::CY1999.Q4    2177.89          2587.76
...
```

Example 6–14 uses the same `unitPrice`, `quarterLevel`, and `prodSel` objects as Example 6–13, but it uses the `unitPriceMovingTotal` measure as the second element of `measuresDim`. The `unitPriceMovingTotal Source` is produced by the `movingTotal` method of `unitPrice`. That method provides `mdmTimeHier`, which is an `MdmLevelHierarchy` component of the Time dimension, as the `dimension` parameter and the integers `0` and `3` as the starting and ending offset values.

The `movingTotalResult Source` specifies, for each quarter, the aggregate of the unit prices for the members of the Desktop PC family for that quarter and the total of that unit price plus the unit prices for the next three quarters.

Example 6–14 Using the movingTotal Method

```
NumberSource unitPriceMovingTotal =
                            unitPrice.movingTotal(mdmTimeHier, 0, 3);

Source measuresDim = dp.createListSource(new Source[]
                                                {unitPrice,
                                                 unitPriceMovingTotal});

Source movingTotalResult = measuresDim.extract()
                                      .join(measuresDim)
                                      .join(quarterLevel)
                                      .joinHidden(prodSel);
```

A `Cursor` for the `movingTotalResult Source` has the following values, displayed in a table with column headings and formatting added. The left column has the quarter, the middle column has the total of the unit prices for the members of the Desktop PC family for that quarter, and the left column has the total of the unit prices for that quarter and the next three quarters. The table includes only the first eight quarters.

```
                                                Unit Price Moving Total
Quarter                            Unit Price  Current Plus Next Three Periods
--------------------------------   ----------  ------------------------------
CALENDAR_YEAR::QUARTER::CY1998.Q1    2687.54           10653.05
CALENDAR_YEAR::QUARTER::CY1998.Q2    2704.48           10360.30
CALENDAR_YEAR::QUARTER::CY1998.Q3    2673.27            9993.00
CALENDAR_YEAR::QUARTER::CY1998.Q4    2587.76            9668.12
CALENDAR_YEAR::QUARTER::CY1999.Q1    2394.79            9258.25
CALENDAR_YEAR::QUARTER::CY1999.Q2    2337.18            8911.87
CALENDAR_YEAR::QUARTER::CY1999.Q3    2348.39            8626.48
CALENDAR_YEAR::QUARTER::CY1999.Q4    2177.89            8291.37
...
```

Selecting a Set of Elements Using Parameterized Source Objects

Example 6–15 uses `NumberParameter` objects to create parameterized `Source` objects. Those objects are the `bottom` and `top` parameters for the `interval` method of `prodHier`. That method produces `paramProdSelInterval`, which is a `Source` that specifies the set of elements of `prodHier` from the `bottom` to the `top` positions of the hierarchy.

The product elements specify the elements of the `units` measure that appear in the `result Source`. By changing the values of the `Parameter` objects, you can select a different set of units sold values using the same `Cursor` and without having to produce new `Source` and `Cursor` objects.

The example uses the following objects.

- `dp`, which is the `DataProvider` for the session.
- `prodHier`, which is the `Source` for the Product Primary hierarchy of the Product dimension.
- `prodShortDescr`, which is the `Source` for the short value description attribute of the Product dimension.
- `units`, which is the `Source` for the Units measure of product units sold.
- `chanHier`, which is the `Source` for the Channel Primary hierarchy of the Channel dimension.
- `calendar`, which is the `Source` for the Calendar Year hierarchy of the Time dimension.
- `shipHier`, which is the `Source` for the Shipments hierarchy of the Customer dimension.
- The `Context11g` object that is returned by the `getContext` method. The `Context11g` has methods that commit the current `Transaction`, that create a `Cursor` for a `Source`, that display text, and that display the values of the `Cursor`.

The `join` method of `prodShortDescr` gets the short value descriptions for the elements of `paramProdSelInterval`. The next four `join` methods match `Source` objects to the inputs of the `units` measure. The example creates a `Cursor` and displays the result set of the query. Next, the `setPosition` method of `resultCursor` sets the position of the `Cursor` back to the first element.

The `setValue` methods of the `NumberParameter` objects change the values of those objects, which changes the selection of product elements specified by the query. The example then displays the values of the `Cursor` again.

Example 6–15 Selecting a Range With NumberParameter Objects

```
NumberParameter startParam = new NumberParameter(dp, 1);
NumberParameter endParam = new NumberParameter(dp, 6);

NumberSource startParamSrc = (NumberSource) startParam.createSource();
NumberSource endParamSrc = (NumberSource) endParam.createSource();

Source paramProdSelInterval = prodHier.interval(startParamSrc,
                                                endParamSrc);
Source paramProdSelIntervalShortDescr =
                                  prodShortDescr.join(paramProdSelInterval);

NumberSource result = (NumberSource)
                       units.join(chanHier, "CHANNEL_PRIMARY::CHANNEL::INT")
                            .join(calendar, "CALENDAR_YEAR::YEAR::CY2001")
                            .join(shipHier, "SHIPMENTS::TOTAL_CUSTOMER::TOTAL")
                            .join(paramProdSelIntervalShortDescr);

// Commit the current transaction.
getContext().commit();
```

```
CursorManager  cursorMngr = dp.createCursorManager(result);
Cursor resultCursor = cursorMngr.createCursor();

getContext().displayCursor(resultCursor);

//Reset the Cursor position to 1;
resultCursor.setPosition(1);

// Change the value of the parameterized Source
startParam.setValue(7);
endParam.setValue(12);

// Display the results again.
getContext().displayCursor(resultsCursor);
```

The following table displays the values of `resultCursor`, with column headings and formatting added. The left column has the product hierarchy elements, the middle column has the short value description, and the right column has the quantity of units sold.

```
              Product                         Description        Units Sold
-------------------------------------  ----------------------  ----------
PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL   Total Product             55,872
PRODUCT_PRIMARY::CLASS::HRD             Hardware                  21,301
PRODUCT_PRIMARY::FAMILY::DISK           Memory                     6,634
PRODUCT_PRIMARY::ITEM::EXT CD ROM       External 48X CD-ROM          136
PRODUCT_PRIMARY::ITEM::EXT DVD          External - DVD-RW - 8X     1,526
PRODUCT_PRIMARY::ITEM::INT 8X DVD       Internal - DVD-RW - 8X     1,543

              Product                         Description        Units Sold
-------------------------------------  ----------------------  ----------
PRODUCT_PRIMARY::ITEM::INT CD ROM       Internal 48X CD-ROM          380
PRODUCT_PRIMARY::ITEM::INT CD USB       Internal 48X CD-ROM USB      162
PRODUCT_PRIMARY::ITEM::INT RW DVD       Internal - DVD-RW - 6X     2,887
PRODUCT_PRIMARY::FAMILY::DTPC           Desktop PCs                2,982
PRODUCT_PRIMARY::ITEM::SENT FIN         Sentinel Financial         1,015
PRODUCT_PRIMARY::ITEM::SENT MM          Sentinel Multimedia          875
```

7

Using a TransactionProvider

This chapter describes the Oracle OLAP Java API Transaction and TransactionProvider interfaces and describes how you use implementations of those interfaces in an application. You get a TransactionProvider from a DataProvider. You use the commitCurrentTransaction method of the TransactionProvider to save a metadata object in persistent storage in the database. You also use that method after creating a derived Source and before creating a Cursor for the Source. For examples of committing a Transaction after creating a metadata object, see Chapter 4.

This chapter includes the following topics:

- About Creating a Metadata Object or a Query in a Transaction
- Using TransactionProvider Objects

About Creating a Metadata Object or a Query in a Transaction

The Oracle OLAP Java API is transactional. Creating metadata objects or Source objects for a query occurs in the context of a Transaction. A TransactionProvider provides Transaction objects to the application and commits or discards those Transaction objects.

The TransactionProvider ensures the following:

- A Transaction is isolated from other Transaction objects. Operations performed in a Transaction are not visible in, and do not affect, other Transaction objects.
- If an operation in a Transaction fails, then the effects of the operation are undone (the Transaction is rolled back).
- The effects of a completed Transaction persist.

When you create a DataProvider and UserSession, the session does not at first have a Transaction. The first Transaction in a session is a root Transaction. You can explicitly create a root Transaction by calling the createRootTransaction method of the TransactionProvider. If you do not explicitly created one, then Oracle OLAP automatically creates a root Transaction the first time that you create or modify an MdmObject or a derived Source. To make permanent the changes to an MdmObject, you must commit the root Transaction in which you made the changes.

A single-user application does not need to explicitly create a root Transaction. The ability to create multiple root Transaction objects is provided for use by multithreaded, middle-tier applications. If your application uses multiple root Transaction objects, the changes that the application makes in one root

Transaction can be overwritten by changes the application makes in another root Transaction. The changes that occur in the last root Transaction that the application commits are the changes that persist.

When you or Oracle OLAP creates the initial root Transaction, it is the *current* Transaction. If you create another root Transaction, it becomes the current Transaction.

Oracle OLAP creates other Transaction objects as you create Source objects or child Transaction objects under a root Transaction. You must commit the root Transaction for the Oracle Database to add to persistent storage any metadata objects that you have created in any Transaction in the session.

When you create a derived Source by calling a method of another Source, the derived Source is created in the context of the current Transaction. The Source is *active* in the Transaction in which you create it or in a child Transaction of that Transaction.

You get or set the current Transaction, or begin a child Transaction, by calling methods of a TransactionProvider. In a child Transaction you can alter a query, for example by changing the selection of dimension elements or by performing a different mathematical or analytical operation on the data, which changes the state of a Template that you created in the parent Transaction. By displaying the data specified by the Source produced by the Template in the parent Transaction and also displaying the data specified by the Source produced by the Template in the child Transaction, you can provide the end user of your application with the means of easily altering a query and viewing the results of different operations on the same set of data, or the same operations on different sets of data.

Types of Transaction Objects

The OLAP Java API has the following two types of Transaction objects:

- A read Transaction. Initially, the current Transaction is a read Transaction. A read Transaction is required for creating a Cursor to fetch data from Oracle OLAP. For more information on Cursor objects, see Chapter 9.
- A write Transaction. A write Transaction is required for creating a derived Source or for changing the state of a Template. For more information on creating a derived Source, see Chapter 5. For information on Template objects, see Chapter 10.

In the initial read Transaction, if you create a derived Source or if you change the state of a Template object, then a child write Transaction is automatically generated. That child Transaction becomes the current Transaction.

If you then create another derived Source or change the Template state again, then that operation occurs in the same write Transaction. You can create any number of derived Source objects, or make any number of Template state changes, in that same write Transaction. You can use those Source objects, or the Source produced by the Template, to define a complex query.

Before you can create a Cursor to fetch the result set specified by a derived Source, you must move the Source from the child write Transaction into the parent read Transaction. To do so, you commit the Transaction.

Committing a Transaction

To move a Source that you created in a child Transaction into the parent read Transaction, call the commitCurrentTransaction method of the

TransactionProvider. When you commit a child write Transaction, a Source you created in the child Transaction moves into the parent read Transaction. The child Transaction disappears and the parent Transaction becomes the current Transaction. The Source is active in the current read Transaction and you can therefore create a Cursor for it.

In Example 7–1, commit() is a method that commits the current Transaction. In the example, dp is the DataProvider.

Example 7–1 Committing the Current Transaction

```
private void commit()
{
  try
  {
    (dp.getTransactionProvider()).commitCurrentTransaction();
  }
  catch (Exception ex)
  {
    System.out.println("Could not commit the Transaction. " + ex);
  }
}
```

About Transaction and Template Objects

Getting and setting the current Transaction, beginning a child Transaction, and rolling back a Transaction are operations that you use to allow an end user to make different selections starting from a given state of a dynamic query.

To present the end user with alternatives based on the same initial query, you do the following:

1. Create a Template in a parent Transaction and set the initial state for the Template.
2. Get the Source produced by the Template, create a Cursor to retrieve the result set, get the values from the Cursor, and then display the results to the end user.
3. Begin a child Transaction and modify the state of the Template.
4. Get the Source produced by the Template in the child Transaction, create a Cursor, get the values, and display them.

You can then replace the first Template state with the second one or discard the second one and retain the first.

Beginning a Child Transaction

To begin a child read Transaction, call the beginSubtransaction method of the TransactionProvider you are using. In the child read Transaction, if you change the state of a Template, then a child write Transaction begins automatically. The write Transaction is a child of the child read Transaction.

To get the data specified by the Source produced by the Template, you commit the write Transaction into the parent read Transaction. You can then create a Cursor to fetch the data. The changed state of the Template is not visible in the original parent. The changed state does not become visible in the parent until you commit the child read Transaction into the parent read Transaction.

After beginning a child read Transaction, you can begin a child read Transaction of that child, or a grandchild of the initial parent Transaction. For an example of creating child and grandchild Transaction objects, see Example 7–3.

About Rolling Back a Transaction

You roll back, or undo, a Transaction by calling the rollbackCurrentTransaction method of the TransactionProvider you are using. Rolling back a Transaction discards any changes that you made during that Transaction and makes the Transaction disappear.

Before rolling back a Transaction, you must close any CursorManager objects you created in that Transaction. After rolling back a Transaction, any Source objects that you created or Template state changes that you made in the Transaction are no longer valid. Any Cursor objects you created for those Source objects are also invalid.

Once you roll back a Transaction, you cannot commit that Transaction. Likewise, once you commit a Transaction, you cannot roll it back.

Example 7–2 Rolling Back a Transaction

The following example uses the TopBottomTemplate and SingleSelectionTemplate classes that are described in Chapter 10, "Creating Dynamic Queries". In creating the TopBottomTemplate and SingleSelectionTemplate objects, the example uses the same code that appears in Example 10–4, "Getting the Source Produced by the Template". Example 7–2 does not show that code. This example sets the state of the TopBottomTemplate. It begins a child Transaction that sets a different state for the TopBottomTemplate and then rolls back the child Transaction. The println method displays text through a CursorPrintWriter object and the getContext method gets a Context11g object that has methods that create Cursor objects and display their values through the CursorPrintWriter. The CursorPrintWriter and Context11g classes are used by the example programs in this documentation.

```
// The current Transaction is a read Transaction, t1.
// Create a TopBottomTemplate using a hierarchy of the Product dimension
// as the base and dp as the DataProvider.
TopBottomTemplate topNBottom = new TopBottomTemplate(prodHier, dp);

// Changing the state of a Template requires a write Transaction, so a
// write child Transaction, t2, is automatically started.
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);
topNBottom.setN(10);
topNBottom.setCriterion(singleSelections.getSource());

// Get the TransactionProvider and commit the Transaction t2.
TransactionProvider tp = dp.getTransactionProvider();
try
{
  tp.commitCurrentTransaction();           // t2 disappears
}
catch(Exception e)
{
  println("Cannot commit the Transaction. " + e);
}

// The current Transaction is now t1.
// Get the dynamic Source produced by the TopBottomTemplate.
```

```
Source result = topNBottom.getSource();

// Create a Cursor and display the results
println("\nThe current state of the TopBottomTemplate" +
        "\nproduces the following values:\n");
getContext().displayTopBottomResult(result);

// Start a child Transaction, t3. It is a read Transaction.
tp.beginSubtransaction();          // t3 is the current Transaction

// Change the state of topNBottom. Changing the state requires a
// write Transaction so Transaction t4 starts automatically.
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);
topNBottom.setN(15);

// Commit the Transaction.
try
{
  tp.commitCurrentTransaction();           // t4 disappears
}
catch(Exception e)
{
  println("Cannot commit the Transaction. " + e);
}

// Create a Cursor and display the results. // t3 is the current Transaction
println("\nIn the child Transaction, the state of the" +
        "\nTopBottomTemplate produces the following values:\n");
getContext().displayTopBottomResult(result);
// The displayTopBottomResult method closes the CursorManager for the
// Cursor created in t3.

// Undo t3, which discards the state of topNBottom that was set in t4.
tp.rollbackCurrentTransaction();        // t3 disappears

// Transaction t1 is now the current Transaction and the state of
// topNBottom is the one defined in t2.

// To show the current state of the TopNBottom template Source, commit
// the Transaction, create a Cursor, and display the Cursor values.
try
{
  tp.commitCurrentTransaction();
}
catch(Exception e)
{
  println("Cannot commit the Transaction. " + e);
}

println("\nAfter rolling back the child Transaction, the state of"
        + "\nthe TopBottomTemplate produces the following values:\n");
getContext().displayTopBottomResult(result);
```

Example 7–2 produces the following output.

```
The current state of the TopBottomTemplate
produces the following values:

 1. PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL
 2. PRODUCT_PRIMARY::CLASS::SFT
 3. PRODUCT_PRIMARY::FAMILY::ACC
```

```
 4. PRODUCT_PRIMARY::CLASS::HRD
 5. PRODUCT_PRIMARY::FAMILY::MOD
 6. PRODUCT_PRIMARY::FAMILY::OS
 7. PRODUCT_PRIMARY::FAMILY::DISK
 8. PRODUCT_PRIMARY::ITEM::MOUSE PAD
 9. PRODUCT_PRIMARY::ITEM::OS 1 USER
10. PRODUCT_PRIMARY::ITEM::DLX MOUSE

In the child Transaction, the state of the
TopBottomTemplate produces the following values:

 1. PRODUCT_PRIMARY::ITEM::EXT CD ROM
 2. PRODUCT_PRIMARY::ITEM::OS DOC ITA
 3. PRODUCT_PRIMARY::ITEM::OS DOC SPA
 4. PRODUCT_PRIMARY::ITEM::INT CD USB
 5. PRODUCT_PRIMARY::ITEM::ENVY EXT KBD
 6. PRODUCT_PRIMARY::ITEM::19 SVGA
 7. PRODUCT_PRIMARY::ITEM::OS DOC FRE
 8. PRODUCT_PRIMARY::ITEM::OS DOC GER
 9. PRODUCT_PRIMARY::ITEM::ENVY ABM
10. PRODUCT_PRIMARY::ITEM::INT CD ROM
11. PRODUCT_PRIMARY::ITEM::ENVY EXE
12. PRODUCT_PRIMARY::ITEM::OS DOC KAN
13. PRODUCT_PRIMARY::ITEM::ENVY STD
14. PRODUCT_PRIMARY::ITEM::1GB USB DRV
15. PRODUCT_PRIMARY::ITEM::SENT MM

After rolling back the child Transaction, the state of
the TopBottomTemplate produces the following values:

 1. PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL
 2. PRODUCT_PRIMARY::CLASS::SFT
 3. PRODUCT_PRIMARY::FAMILY::ACC
 4. PRODUCT_PRIMARY::CLASS::HRD
 5. PRODUCT_PRIMARY::FAMILY::MOD
 6. PRODUCT_PRIMARY::FAMILY::OS
 7. PRODUCT_PRIMARY::FAMILY::DISK
 8. PRODUCT_PRIMARY::ITEM::MOUSE PAD
 9. PRODUCT_PRIMARY::ITEM::OS 1 USER
10. PRODUCT_PRIMARY::ITEM::DLX MOUSE
```

Getting and Setting the Current Transaction

You get the current `Transaction` by calling the `getCurrentTransaction` method of the `TransactionProvider` you are using, as in the following example.

```
Transaction t1 = tp.getCurrentTransaction();
```

To make a previously saved `Transaction` the current `Transaction`, you call the `setCurrentTransaction` method of the `TransactionProvider`, as in the following example.

```
tp.setCurrentTransaction(t1);
```

Using TransactionProvider Objects

In the Oracle OLAP Java API, a `DataProvider` provides an implementation of the `TransactionProvider` interface. The `TransactionProvider` provides `Transaction` objects to your application.

As described in "Committing a Transaction" on page 7-2, you use the commitCurrentTransaction method to make a derived Source that you created in a child write Transaction visible in the parent read Transaction. You can then create a Cursor for that Source.

If you are using Template objects in your application, then you might also use the other methods of TransactionProvider to do the following:

- Begin a child Transaction.
- Get the current Transaction so you can save it.
- Set the current Transaction to a previously saved one.
- Rollback, or undo, the current Transaction, which discards any changes made in the Transaction. Once a Transaction has been rolled back, it is invalid and cannot be committed. Once a Transaction has been committed, it cannot be rolled back. If you created a Cursor for a Source in a Transaction, then you must close the CursorManager before rolling back the Transaction.

Example 7–3 demonstrates the use of Transaction objects to modify dynamic queries. Like Example 7–2, this example uses the same code to create TopBottomTemplate and SingleSelectionTemplate objects as does Example 10–4, "Getting the Source Produced by the Template". This example does not show that code.

To help track the Transaction objects, this example saves the different Transaction objects with calls to the getCurrentTransaction method. In the example, tp object is the TransactionProvider. The println method displays text through a CursorPrintWriter and the getContext method gets a Context11g object that has methods that create Cursor objects and display their values through the CursorPrintWriter. The commit method is the method from Example 7–1.

Example 7–3 Using Child Transaction Objects

```
// The parent Transaction is the current Transaction at this point.
// Save the parent read Transaction as parentT1.
Transaction parentT1 = tp.getCurrentTransaction();

// Get the dynamic Source produced by the TopBottomTemplate.
Source result = topNBottom.getSource();

// Create a Cursor and display the results.
println("\nThe current state of the TopBottomTemplate" +
        "\nproduces the following values:\n");
getContext().displayTopBottomResult(result);

// Begin a child Transaction of parentT1.
tp.beginSubtransaction();  // This is a read Transaction.

// Save the child read Transaction as childT2.
Transaction childT2 = tp.getCurrentTransaction();

// Change the state of the TopBottomTemplate. This starts a
// write Transaction, a child of the read Transaction childT2.
topNBottom.setN(12);
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);

// Save the child write Transaction as writeT3.
Transaction writeT3 = tp.getCurrentTransaction();
```

```
// Commit the write Transaction writeT3.
commit();

// The commit moves the changes made in writeT3 into its parent,
// the read Transaction childT2. The writeT3 Transaction
// disappears. The current Transaction is now childT2
// again but the state of the TopBottomTemplate has changed.

// Create a Cursor and display the results of the changes to the
// TopBottomTemplate that are visible in childT2.
try
{
  println("\nIn the child Transaction, the state of the" +
          "\nTopBottomTemplate produces the following values:\n");

  getContext().displayTopBottomResult(result);
}
catch(Exception e)
{
  println("Cannot display the results of the query. " + e);
}

// Begin a grandchild Transaction of the initial parent.
tp.beginSubtransaction();  // This is a read Transaction.

// Save the grandchild read Transaction as grandchildT4.
Transaction grandchildT4 = tp.getCurrentTransaction();

// Change the state of the TopBottomTemplate. This starts another
// write Transaction, a child of grandchildT4.
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);

// Save the write Transaction as writeT5.
Transaction writeT5 = tp.getCurrentTransaction();

// Commit writeT5.
commit();

// Transaction grandchildT4 is now the current Transaction and the
// changes made to the TopBottomTemplate state are visible.

// Create a Cursor and display the results visible in grandchildT4.
try
{
  println("\nIn the grandchild Transaction, the state of the" +
          "\nTopBottomTemplate produces the following values:\n");
  getContext().displayTopBottomResult(result);
}
catch(Exception e)
{
  println(""Cannot display the results of the query. " + e);
}

// Commit the grandchild into the child.
commit();

// Transaction childT2 is now the current Transaction.
// Instead of preparing and committing the grandchild Transaction,
// you could rollback the Transaction, as in the following
```

```
// method call:
//   rollbackCurrentTransaction();
// If you roll back the grandchild Transaction, then the changes
// you made to the TopBottomTemplate state in the grandchild
// are discarded and childT2 is the current Transaction.

// Commit the child into the parent.
commit();

// Transaction parentT1 is now the current Transaction. Again,
// you can roll back the childT2 Transaction instead of committing it.
// If you do so, then the changes that you made in childT2 are discarded.
// The current Transaction is be parentT1, which has the original state
// of the TopBottomTemplate, without any of the changes made in the
// grandchild or the child transactions.
```

Example 7–3 produces the following output.

```
The current state of the TopBottomTemplate
produces the following values:

 1. PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL
 2. PRODUCT_PRIMARY::CLASS::SFT
 3. PRODUCT_PRIMARY::FAMILY::ACC
 4. PRODUCT_PRIMARY::CLASS::HRD
 5. PRODUCT_PRIMARY::FAMILY::MOD
 6. PRODUCT_PRIMARY::FAMILY::OS
 7. PRODUCT_PRIMARY::FAMILY::DISK
 8. PRODUCT_PRIMARY::ITEM::MOUSE PAD
 9. PRODUCT_PRIMARY::ITEM::OS 1 USER
10. PRODUCT_PRIMARY::ITEM::DLX MOUSE

In the child Transaction, the state of the
TopBottomTemplate produces the following values:

 1. PRODUCT_PRIMARY::ITEM::EXT CD ROM
 2. PRODUCT_PRIMARY::ITEM::OS DOC ITA
 3. PRODUCT_PRIMARY::ITEM::OS DOC SPA
 4. PRODUCT_PRIMARY::ITEM::INT CD USB
 5. PRODUCT_PRIMARY::ITEM::ENVY EXT KBD
 6. PRODUCT_PRIMARY::ITEM::19 SVGA
 7. PRODUCT_PRIMARY::ITEM::OS DOC FRE
 8. PRODUCT_PRIMARY::ITEM::OS DOC GER
 9. PRODUCT_PRIMARY::ITEM::ENVY ABM
10. PRODUCT_PRIMARY::ITEM::INT CD ROM
11. PRODUCT_PRIMARY::ITEM::ENVY EXE
12. PRODUCT_PRIMARY::ITEM::OS DOC KAN

In the grandchild Transaction, the state of the
TopBottomTemplate produces the following values:

 1. PRODUCT_PRIMARY::TOTAL_PRODUCT::TOTAL
 2. PRODUCT_PRIMARY::CLASS::SFT
 3. PRODUCT_PRIMARY::FAMILY::ACC
 4. PRODUCT_PRIMARY::CLASS::HRD
 5. PRODUCT_PRIMARY::FAMILY::MOD
 6. PRODUCT_PRIMARY::FAMILY::OS
 7. PRODUCT_PRIMARY::FAMILY::DISK
 8. PRODUCT_PRIMARY::ITEM::MOUSE PAD
 9. PRODUCT_PRIMARY::ITEM::OS 1 USER
```

```
10. PRODUCT_PRIMARY::ITEM::DLX MOUSE
11. PRODUCT_PRIMARY::ITEM::LT CASE
12. PRODUCT_PRIMARY::ITEM::56KPS MODEM
```

8

Understanding Cursor Classes and Concepts

This chapter describes the Oracle OLAP Java API `Cursor` class and its related classes, which you use to retrieve the results of a query. This chapter also describes the `Cursor` concepts of position, fetch size, and extent. For examples of creating and using a `Cursor` and its related objects, see Chapter 9, "Retrieving Query Results".

This chapter includes the following topics:

Overview of the OLAP Java API Cursor Objects

A `Cursor` retrieves the result set defined by a `Source`. You create a `Cursor` by calling the `createCursor` method of a `CursorManager`. You create a `CursorManager` by calling the `createCursorManager` method of a `DataProvider`.

You can get the SQL generated for a `Source` by the Oracle OLAP SQL generator without having to create a `Cursor`. To get the SQL for the `Source`, you create an `SQLCursorManager` by using a `createSQLCursorManager` method of a `DataProvider`. You can then use classes outside of the OLAP Java API, or other methods, to retrieve data using the generated SQL.

Creating a Cursor

You create a `Cursor` for a `Source` by doing the following:

1. Creating a `CursorManager` by calling one of the `createCursorManager` methods of the `DataProvider` and passing it the `Source`. If you want to alter the behavior of the `Cursor`, then you can create a `CursorInfoSpecification` and use the methods of it to specify the behavior. You then create a `CursorManager` with a method that takes the `Source` and the `CursorInfoSpecification`.

2. Creating a `Cursor` by calling the `createCursor` method of the `CursorManager`.

Sources For Which You Cannot Create a Cursor

Some Source objects do not specify data that a Cursor can retrieve from the data store. The following are Source objects for which you cannot create a Cursor that contains values.

- A Source that specifies an operation that is not computationally possible. An example is a Source that specifies an infinite recursion.
- A Source that defines an infinite result set. An example is the fundamental Source that represents the set of all String objects.
- A Source that has no elements or includes another Source that has no elements. Examples are a Source returned by the getEmptySource method of DataProvider and another Source derived from the empty Source. Another example is a derived Source that results from selecting a value from a primary Source that you got from an MdmDimension and the selected value does not exist in the dimension.

If you create a Cursor for such a Source and try to get the values of the Cursor, then an Exception occurs.

Cursor Objects and Transaction Objects

When you create a derived Source or change the state of a Template, you create the Source in the context of the current Transaction. The Source is active in the Transaction in which you create it or in a child Transaction of that Transaction. A Source must be active in the current Transaction for you to be able to create a Cursor for it.

Creating a derived Source occurs in a write Transaction. Creating a Cursor occurs in a read Transaction. After creating a derived Source, and before you can create a Cursor for that Source, you must change the write Transaction into a read Transaction by calling the commitCurrentTransaction methods of the TransactionProvider your application is using. For information on Transaction and TransactionProvider objects, see Chapter 7, "Using a TransactionProvider".

For a Cursor that you create for a query that includes a parameterized Source, you can change the value of the Parameter object and then get the new values of the Cursor without having to commit the Transaction again. For information on parameterized Source objects, see Chapter 5, "Understanding Source Objects".

Cursor Classes

In the oracle.olapi.data.cursor package, the Oracle OLAP Java API defines the interfaces described in the following table.

Interface	Description
Cursor	An abstract superclass that encapsulates the notion of a current position.
ValueCursor	A Cursor that has a value at the current position. A ValueCursor has no child Cursor objects.
CompoundCursor	A Cursor that has child Cursor objects, which are a child ValueCursor for the values of the Source associated with it and an output child Cursor for each output of the Source.

Structure of a Cursor

The structure of a Cursor mirrors the structure of the Source associated with it. If the Source does not have any outputs, then the Cursor for that Source is a ValueCursor. If the Source has one or more outputs, then the Cursor for that Source is a CompoundCursor. A CompoundCursor has as children a base ValueCursor, which has the values of the base of the Source of the CompoundCursor, and one or more output Cursor objects.

The output of a Source is another Source. An output Source can itself have outputs. The child Cursor for an output of a Source is a ValueCursor if the output Source does not have any outputs and a CompoundCursor if it does.

Example 8–1 creates a query that specifies the prices of selected product items for selected months. In the example, timeHier is a Source for a hierarchy of a dimension of time values, and prodHier is a Source for a hierarchy of a dimension of product values.

If you create a Cursor for prodSel or for timeSel, then either Cursor is a ValueCursor because both prodSel and timeSel have no outputs.

The unitPrice object is a Source for an MdmMeasure that represents values for the price of product units. The MdmMeasure has as inputs the MdmPrimaryDimension objects representing products and times, and the unitPrice Source has as inputs the Source objects for those dimensions.

The example selects elements of the dimension hierarchies and then joins the Source objects for the selections to that of the measure to produce querySource, which has prodSel and timeSel as outputs.

Example 8–1 Creating the querySource Query

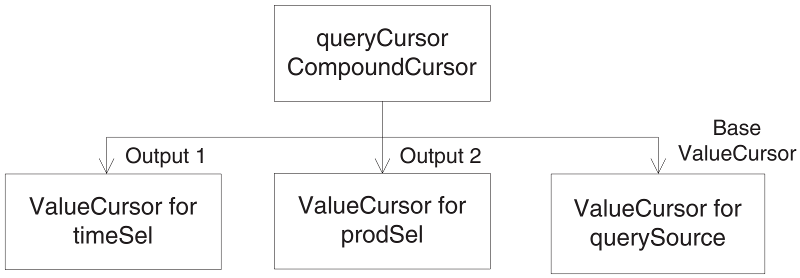
```
Source timeSel = timeHier.selectValues(new String[]
                                       {"CALENDAR_YEAR::MONTH::2001.01",
                                        "CALENDAR_YEAR::MONTH::2001.04",
                                        "CALENDAR_YEAR::MONTH::2001.07",
                                        "CALENDAR_YEAR::MONTH::2001.10"});

Source prodSel = prodHier.selectValues(new String[]
                                       {"PRODUCT_PRIMARY::ITEM::ENVY ABM",
                                        "PRODUCT_PRIMARY::ITEM::ENVY EXE",
                                        "PRODUCT_PRIMARY::ITEM::ENVY STD"});

Source querySource = unitPrice.join(timeSel).join(prodSel);
```

The result set defined by querySource is the unit price values for the selected products for the selected months. The results are organized by the outputs. Since timeSel is joined to the Source produced by the unitPrice.join(prodSel) operation, timeSel is the slower varying output, which means that the result set specifies the set of selected products for each selected time value. For each time value the result set has three product values so the product values vary faster than the time values. The values of the base ValueCursor of querySource are the fastest varying of all, because there is one price value for each product for each day.

Example 9–1 in Chapter 9, creates a Cursor, queryCursor, for querySource. Since querySource has outputs, queryCursor is a CompoundCursor. The base ValueCursor of queryCursor has values from unitPrice, which is the base Source of the operation that created querySource. The values from unitPrice are those specified by the outputs.The outputs for queryCursor are a ValueCursor that has values from prodSel and a ValueCursor that has values from timeSel.

Figure 8–1 illustrates the structure of `queryCursor`. The base `ValueCursor` and the two output `ValueCursor` objects are the children of `queryCursor`, which is the parent `CompoundCursor`.

Figure 8–1 Structure of the queryCursor CompoundCursor



The following table displays the values from `queryCursor` in a table. The left column has time values, the middle column has product values, and the right column has the unit price of the product for the month.

Month	Product	Price of Unit
2001.01	ENVY ABM	3042.22
2001.01	ENVY EXE	3223.28
2001.01	ENVY STD	3042.22
2001.04	ENVY ABM	2412.42
2001.04	ENVY EXE	3107.65
2001.04	ENVY STD	3026.12
2001.07	ENVY ABM	2505.57
2001.07	ENVY EXE	3155.91
2001.07	ENVY STD	2892.18
2001.10	ENVY ABM	2337.30
2001.10	ENVY EXE	3105.53
2001.10	ENVY STD	2856.86

For examples of getting the values from a `ValueCursor`, see Chapter 9.

Specifying the Behavior of a Cursor

`CursorSpecification` objects specify some aspects of the behavior of their corresponding `Cursor` objects. You must specify the behavior on a `CursorSpecification` before creating the corresponding `Cursor`. To specify the behavior, use the following `CursorSpecification` methods:

- `setDefaultFetchSize`
- `setExtentCalculationSpecified`
- `setParentEndCalculationSpecified`
- `setParentStartCalculationSpecified`

- `specifyDefaultFetchSizeOnChildren` (for a `CompoundCursorSpecification` only)

A `CursorSpecification` also has methods that you can use to discover if the behavior is specified. Those methods are the following:

- `isExtentCalculationSpecified`
- `isParentEndCalculationSpecified`
- `isParentStartCalculationSpecified`

If you have used the `CursorSpecification` methods to set the default fetch size, or to calculate the extent or the starting or ending positions of a value in the parent of the value, then you can successfully use the following `Cursor` methods:

- `getExtent`
- `getFetchSize`
- `getParentEnd`
- `getParentStart`
- `setFetchSize`

For examples of specifying `Cursor` behavior, see Chapter 9. For information on fetch sizes, see "About Fetch Sizes" on page 8-13. For information on the extent of a `Cursor`, see "What is the Extent of a Cursor?" on page 8-12. For information on the starting and ending positions in a parent `Cursor` of the current value of a `Cursor`, see "About the Parent Starting and Ending Positions in a Cursor" on page 8-12.

CursorInfoSpecification Classes

The `CursorInfoSpecification` interface and the subinterfaces `CompoundCursorInfoSpecification` and `ValueCursorInfoSpecification`, specify methods for the abstract `CursorSpecification` class and the concrete `CompoundCursorSpecification` and `ValueCursorSpecification` classes. A `CursorSpecification` specifies certain aspects of the behavior of the `Cursor` that corresponds to it. You can create instances of classes that implement the `CursorInfoSpecification` interface either directly or indirectly.

You can create a `CursorSpecification` for a `Source` by calling the `createCursorInfoSpecification` method of a `DataProvider`. That method returns a `CompoundCursorSpecification` or a `ValueCursorSpecification`. You can use the methods of the `CursorSpecification` to specify aspects of the behavior of a `Cursor`. You can then use the `CursorSpecification` in creating a `CursorManager` by passing it as the `cursorInfoSpec` argument to the `createCursorManager` method of a `DataProvider`.

With `CursorSpecification` methods, you can do the following:

- Get the `Source` that corresponds to the `CursorSpecification`.
- Get or set the default fetch size for the corresponding `Cursor`.
- Specify that Oracle OLAP should calculate the extent of a `Cursor`.
- Determine whether calculating the extent is specified.
- Specify that Oracle OLAP should calculate the starting or ending position of the current value of the corresponding `Cursor` in the parent `Cursor`. If you know the starting and ending positions of a value in the parent, then you can determine how many faster varying elements the parent `Cursor` has for that value.

- Determine whether calculating the starting or ending position of the current value of the corresponding `Cursor` in the parent is specified.
- Accept a `CursorSpecificationVisitor`.

For more information, see "About Cursor Positions and Extent" on page 8-7 and "About Fetch Sizes" on page 8-13.

In the `oracle.olapi.data.source` package, the Oracle OLAP Java API defines the classes described in the following table.

Interface	Description
`CursorInfoSpecification`	An interface that specifies methods for `CursorSpecification` objects.
`CursorSpecification`	An abstract class that implements some methods of the `CursorInfoSpecification` interface.
`CompoundCursorSpecification`	A `CursorSpecification` for a `Source` that has one or more outputs. A `CompoundCursorSpecification` has component child `CursorSpecification` objects.
`CompoundInfoCursorSpecification`	An interface that specifies methods for `CompoundCursorSpecification` objects.
`ValueCursorSpecification`	A `CursorSpecification` for a `Source` that has values and no outputs.
`ValueCursorInfoSpecification`	An interface for `ValueCursorSpecification` objects.

A `Cursor` has the same structure as the `CursorSpecification`. Every `ValueCursorSpecification` or `CompoundCursorSpecification` has a corresponding `ValueCursor` or `CompoundCursor`. To be able to get certain information or behavior from a `Cursor`, your application must specify that it wants that information or behavior by calling methods of the corresponding `CursorSpecification` before it creates the `Cursor`.

CursorManager Class

With a `CursorManager`, you can create a `Cursor` for a `Source`. The class returned by one of the `createCursorManager` methods of a `DataProvider` manages the buffering of data for the `Cursor` objects it creates.

You can create more than one `Cursor` from the same `CursorManager`, which is useful for displaying data from a result set in different formats such as a table or a graph. All of the `Cursor` objects created by a `CursorManager` have the same specifications, such as the default fetch sizes. Because the `Cursor` objects have the same specifications, they can share the data managed by the `CursorManager`.

A `SQLCursorManager` has methods that return the SQL generated by the Oracle OLAP SQL generator for a `Source`. You create one or more `SQLCursorManager` objects by calling the `createSQLCursorManager` or `createSQLCursorManagers` methods of a `DataProvider`. You do not use a `SQLCursorManager` to create a `Cursor`. Instead, you use the SQL returned by the `SQLCursorManager` with classes outside of the OLAP Java API, or by other means, to retrieve the data specified by the query.

Updating the CursorInfoSpecification for a CursorManager

If your application is using OLAP Java API `Template` objects and the state of a `Template` changes in a way that alters the structure of the `Source` produced by the `Template`, then any `CursorInfoSpecification` objects for the `Source` are no longer valid. You need to create new `CursorInfoSpecification` objects for the changed `Source`.

After creating a new `CursorInfoSpecification`, you can create a new `CursorManager` for the `Source`. You do not, however, need to create a new `CursorManager`. You can call the `updateSpecification` method of the existing `CursorManager` to replace the previous `CursorInfoSpecification` with the new `CursorInfoSpecification`. You can then create a new `Cursor` from the `CursorManager`.

About Cursor Positions and Extent

A `Cursor` has one or more positions. The current position of a `Cursor` is the position that is currently active in the `Cursor`. To move the current position of a `Cursor` call the `setPosition` or `next` methods of the `Cursor`.

Oracle OLAP does not validate the position that you set on the `Cursor` until you attempt an operation on the `Cursor`, such as calling the `getCurrentValue` method. If you set the current position to a negative value or to a value that is greater than the number of positions in the `Cursor` and then attempt a `Cursor` operation, then the `Cursor` throws a `PositionOutOfBoundsException`.

The extent of a `Cursor` is described in "What is the Extent of a Cursor?" on page 8-12.

Positions of a ValueCursor

The current position of a `ValueCursor` specifies a value, which you can retrieve. For example, `prodSel`, a derived `Source` described in "Structure of a Cursor" on page 8-3, is a selection of three products from a primary `Source` that specifies a dimension of products and their hierarchical groupings. The `ValueCursor` for `prodSel` has three elements. The following example gets the position of each element of the `ValueCursor`, and displays the value at that position. The `context` object has a method that displays text.

```
// prodSelValCursor is the ValueCursor for prodSel
println("ValueCursor Position  Value ");
println("--------------------  ------------------------");
do
{
  println("          " + prodSelValCursor.getPosition() +
                  "           " + prodSelValCursor.getCurrentValue());
} while(prodSelValCursor.next());
```

The preceding example displays the following:

```
ValueCursor Position              Value
--------------------  ------------------------------
          1           PRODUCT_PRIMARY::ITEM::ENVY ABM
          2           PRODUCT_PRIMARY::ITEM::ENVY EXE
          3           PRODUCT_PRIMARY::ITEM::ENVY STD
```

The following example sets the current position of prodSelValCursor to 2 and retrieves the value at that position.

```
prodSelValCursor.setPosition(2);
println(prodSelValCursor.getCurrentString());
```

The preceding example displays the following:

```
PRODUCT_PRIMARY::ITEM::ENVY EXE
```

For more examples of getting the current value of a ValueCursor, see Chapter 9.

Positions of a CompoundCursor

A CompoundCursor has one position for each set of the elements of the descendent ValueCursor objects. The current position of the CompoundCursor specifies one of those sets.

For example, querySource, the Source created in Example 8–1, has values from a measure, unitPrice. The values are the prices of product units at different times. The outputs of querySource are Source objects that represent selections of four month values from a time dimension and three product values from a product dimension.

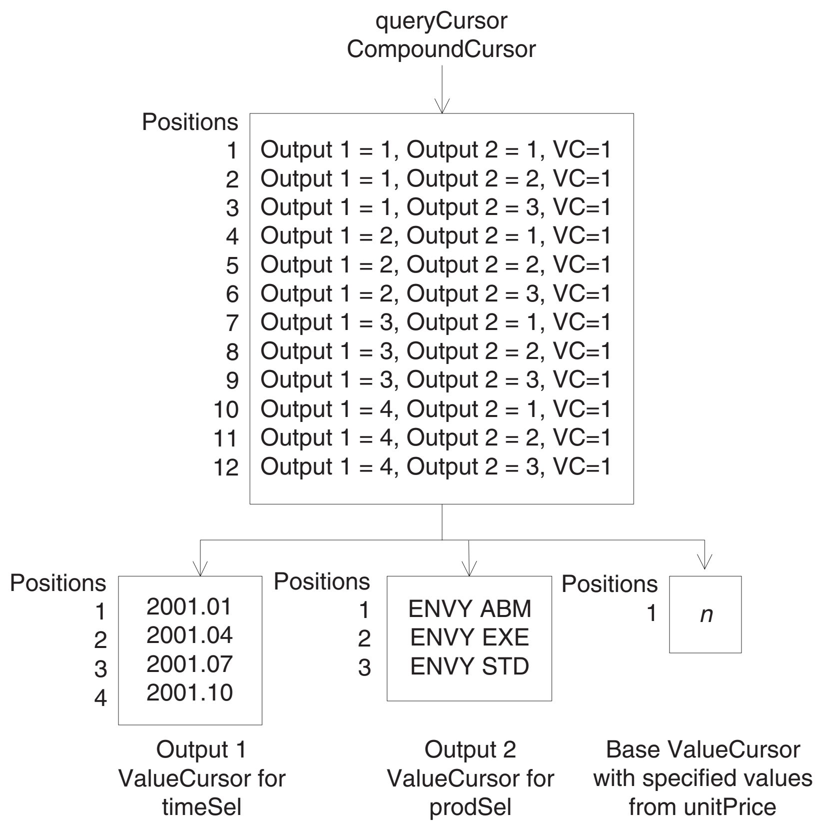
The result set for querySource has one measure value for each tuple (each set of output values), so the total number of values is twelve (one value for each of the three products for each of the four months). Therefore, the queryCursor CompoundCursor created for querySource has twelve positions.

Each position of queryCursor specifies one set of positions of the outputs and the base ValueCursor. For example, position 1 of queryCursor defines the following set of positions for the outputs and the base ValueCursor:

- Position 1 of output 1 (the ValueCursor for timeSel)
- Position 1 of output 2 (the ValueCursor for prodSel)
- Position 1 of the base ValueCursor for queryCursor (This position has the value from the unitPrice measure that is specified by the values of the outputs.)

Figure 8–2 illustrates the positions of queryCursor CompoundCursor, the base ValueCursor, and the outputs.

Figure 8–2 Cursor Positions in queryCursor



The `ValueCursor` for `queryCursor` has only one position because only one value of `unitPrice` is specified by any one set of values of the outputs. For a query such as `querySource`, the `ValueCursor` of the `Cursor` has only one value, and therefore only one position, at a time for any one position of the root `CompoundCursor`.

Figure 8–3 illustrates one possible display of the data from `queryCursor`. It is a crosstab view with four columns and five rows. In the left column are the month values. In the top row are the product values. In each of the intersecting cells of the crosstab is the price of the product for the month.

Figure 8–3 Crosstab Display of queryCursor

	Product		
Month	**ENVY ABM**	**ENVY EXE**	**ENVY STD**
2001.01	3042.22	3223.28	2426.07
2001.04	3026.12	3107.65	2412.42
2001.07	2892.18	3155.91	2505.57
2001.10	2892.18	3105.53	2337.30

A `CompoundCursor` coordinates the positions of the `ValueCursor` objects relative to each other. The current position of the `CompoundCursor` specifies the current positions of the descendent `ValueCursor` objects. Example 8–2 sets the position of queryCursor and then gets the current values and the positions of the child Cursor objects.

Example 8–2 Setting the CompoundCursor Position and Getting the Current Values

```
CompoundCursor rootCursor = (CompoundCursor) queryCursor;
ValueCursor baseValueCursor = rootCursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor output1 = (ValueCursor) outputs.get(0);
ValueCursor output2 = (ValueCursor) outputs.get(1);
int pos = 5;
rootCursor.setPosition(pos);
println("CompoundCursor position set to " + pos + ".");
println("The current position of the CompoundCursor is = " +
        rootCursor.getPosition() + ".");
println("Output 1 position = " + output1.getPosition() +
        ", value = " + output1.getCurrentValue());
println("Output 2 position = " + output2.getPosition() +
        ", value = " + output2.getCurrentValue());
println("VC position = " + baseValueCursor.getPosition() +
        ", value = " + baseValueCursor.getCurrentValue());
```

Example 8–2 displays the following:

```
CompoundCursor position set to 5.
The current position of the CompoundCursor is 5.
Output 1 position = 2, value = CALENDAR_YEAR::MONTH::2001.04
Output 2 position = 2, value = PRODUCT_PRIMARY::ITEM::ENVY EXE
VC position = 1, value = 3107.65
```

The positions of queryCursor are symmetric in that the result set for querySource always has three product values for each time value. The ValueCursor for prodSel, therefore, always has three positions for each value of the timeSel ValueCursor. The timeSel output ValueCursor is slower varying than the prodSel ValueCursor.

In an asymmetric case, however, the number of positions in a ValueCursor is not always the same relative to the slower varying output. For example, if the price of units for product ENVY ABM for month 2001.10 were null because that product was no longer being sold by that date, and if null values were suppressed in the query, then queryCursor would only have eleven positions. The ValueCursor for prodSel would only have two positions when the position of the ValueCursor for timeSel was 4.

Example 8–3 demonstrates an asymmetric result set that is produced by selecting elements of one dimension based on a comparison of measure values. The example uses the same product and time selections as in Example 8–1. It uses a Source for a measure of product units sold, units, that is dimensioned by product, time, sales channels, and customer dimensions. The chanSel and custSel objects are selections of single values of the dimensions. The example produces a Source, querySource2, that specifies which of the selected products sold more than one unit for the selected time, channel, and customer values. Because querySource2 is a derived Source, this example commits the current Transaction.

The example creates a Cursor for querySource2, loops through the positions of the CompoundCursor, gets the position and current value of the first output ValueCursor and the ValueCursor of the CompoundCursor, and displays the positions and values of the ValueCursor objects. The getLocalValue method is a method in the program that extracts the local value from a unique value.

Example 8–3 Positions in an Asymmetric Query

```
// Create the query
prodSel.join(chanSel).join(custSel).join(timeSel).select(units.gt(1));

// Commit the current Transaction.
try
{  // The DataProvider is dp.
  (dp.getTransactionProvider()).commitCurrentTransaction();
}
catch(Exception e)
{
  output.println("Cannot commit current Transaction " + e);
}

// Create the CursorManager and the Cursor.
CursorManager cursorManager = dp.createCursorManager(querySource2);
Cursor queryCursor2 = cursorManager.createCursor();

CompoundCursor rootCursor = (CompoundCursor) queryCursor2;
ValueCursor baseValueCursor = rootCursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor output1 = (ValueCursor) outputs.get(0);

// Get the positions and values and display them.
println("CompoundCursor Output ValueCursor" + "    ValueCursor");
println("  position      position | value  " + "position | value");
do
{
println(sp6 + rootCursor.getPosition() +  // sp6 is 6 spaces
        sp13 + output1.getPosition() +    // sp13 is 13 spaces
        sp7 + getLocalValue(output1.getCurrentString()) + //sp7 is 7 spaces
        sp7 + baseValueCursor.getPosition() +
        sp7 + getLocalValue(baseValueCursor.getCurrentString()));
}
while(queryCursor2.next());
```

Example 8–3 displays the following:

```
CompoundCursor Output ValueCursor       ValueCursor
  position      position |    value    position |    value
      1             1       2001.01       1       ENVY ABM
      2             1       2001.01       2       ENVY EXE
      3             1       2001.01       3       ENVY STD
      4             2       2001.04       1       ENVY ABM
      5             3       2001.07       1       ENVY ABM
      6             3       2001.07       2       ENVY EXE
      7             4       2001.10       1       ENVY EXE
      8             4       2001.10       2       ENVY STD
```

Because not every combination of product and time selections has unit sales greater than 1 for the specified channel and customer selections, the number of elements of the `ValueCursor` for the values derived from `prodSel` is not the same for each value of the output `ValueCursor`. For time value 2001.01, all three products have sales greater than one, but for time value 2001.04, only one of the products does. The other two time values, 2001.07 and 2001.10, have two products that meet the criteria. Therefore, the `ValueCursor` for the `CompoundCursor` has three positions for time 2001.01, only one position for time 2001.04, and two positions for times 2001.07 and 2001.10.

About the Parent Starting and Ending Positions in a Cursor

To effectively manage the display of the data that you get from a CompoundCursor, you sometimes need to know how many faster varying values exist for the current slower varying value. For example, suppose that you are displaying in a crosstab one row of values from an edge of a cube, then you might want to know how many columns to draw in the display for the row.

To determine how many faster varying values exist for the current value of a child Cursor, you find the starting and ending positions of that current value in the parent Cursor. Subtract the starting position from the ending position and then add 1, as in the following.

```
long span = (cursor.getParentEnd() - cursor.getParentStart()) + 1;
```

The result is the span of the current value of the child Cursor in the parent Cursor, which tells you how many values of the fastest varying child Cursor exist for the current value. Calculating the starting and ending positions is costly in time and computing resources, so you should only specify that you want those calculations performed when your application needs the information.

An Oracle OLAP Java API Cursor enables your application to have only the data that it is currently displaying actually present on the client computer. For information on specifying the amount of data for a Cursor, see "About Fetch Sizes" on page 8-13.

From the data on the client computer, however, you cannot determine at what position of the parent Cursor the current value of a child Cursor begins or ends. To get that information, you use the getParentStart and getParentEnd methods of a Cursor.

To specify that you want Oracle OLAP to calculate the starting and ending positions of a value of a child Cursor in the parent Cursor, call the setParentStartCalculationSpecified and setParentEndCalculationSpecified methods of the CursorSpecification corresponding to the Cursor. You can determine whether calculating the starting or ending positions is specified by calling the isParentStartCalculationSpecified or isParentEndCalculationSpecified methods of the CursorSpecification. For an example of specifying these calculations, see Chapter 9.

What is the Extent of a Cursor?

The extent of a Cursor is the total number of elements it contains relative to any slower varying outputs.

The extent is information that you can use, for example, to display the correct number of columns or correctly-sized scroll bars. The extent, however, can be expensive to calculate. For example, a Source that represents a cube might have four outputs. Each output might have hundreds of values. If all null values and zero values of the measure for the sets of outputs are eliminated from the result set, then to calculate the extent of the CompoundCursor for the Source, Oracle OLAP must traverse the entire result space before it creates the CompoundCursor. If you do not specify that you wants the extent calculated, then Oracle OLAP only needs to traverse the sets of elements defined by the outputs of the cube as specified by the fetch size of the Cursor and as needed by your application.

To specify that you want Oracle OLAP to calculate the extent for a Cursor, call the setExtentCalculationSpecified method of the CursorSpecification corresponding to the Cursor. You can determine whether calculating the extent is specified by calling the isExtentCalculationSpecified method of the CursorSpecification. For an example of specifying the calculation of the extent of a Cursor, see Chapter 9.

About Fetch Sizes

An OLAP Java API Cursor represents the entire result set for a Source. The Cursor is a virtual Cursor, however, because it retrieves only a portion of the result set at a time from Oracle OLAP. A CursorManager manages a virtual Cursor and retrieves results from Oracle OLAP as your application needs them. By managing the virtual Cursor, the CursorManager relieves your application of a substantial burden.

The amount of data that a Cursor retrieves in a single fetch operation is determined by the fetch size specified for the Cursor. You specify a fetch size to limit the amount of data your application needs to cache on the local computer and to maximize the efficiency of the fetch by customizing it to meet the needs of your method of displaying the data.

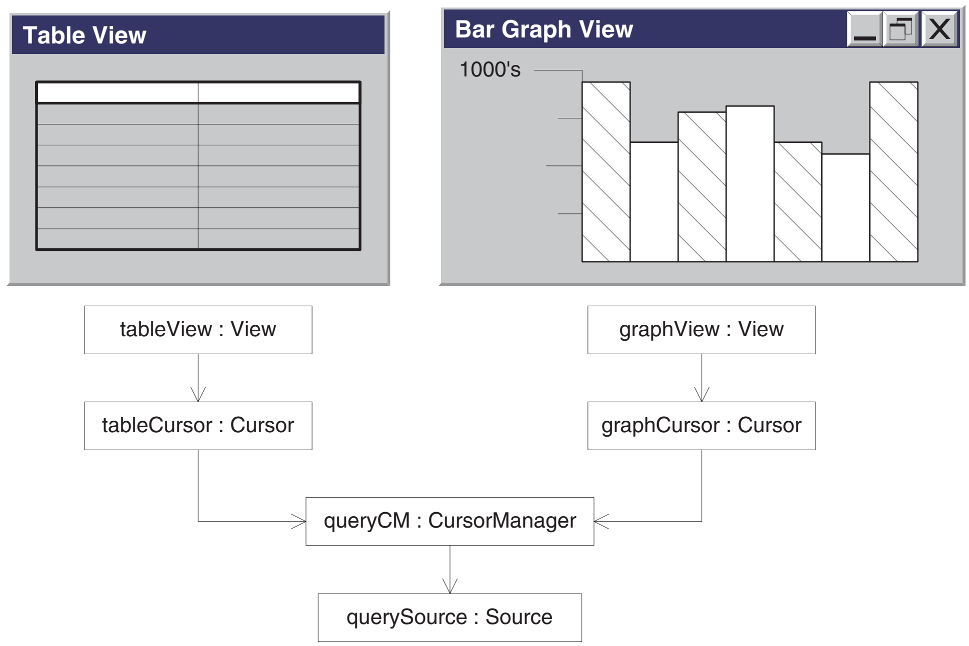
You can also regulate the number of elements that Oracle OLAP returns by using Parameter and parameterized Source objects in constructing your query. For more information on Parameter objects, see Chapter 5, "Understanding Source Objects". For examples of using parameterized Source objects, see Chapter 6, "Making Queries Using Source Methods".

When you create a CursorManager for a Source, Oracle OLAP specifies a default fetch size on the root CursorSpecification. You can change the default fetch size with the setDefaultFetchSize method of the root CursorSpecification.

You can create two or more Cursor objects from the same CursorManager and use both Cursor objects simultaneously. Rather than having separate data caches, the Cursor objects can share the data managed by the CursorManager.

An example is an application that displays the results of a query to the user as both a table and a graph. The application creates a CursorManager for the Source. The application creates two separate Cursor objects from the same CursorManager, one for a table view and one for a graph view. The two views share the same query and display the same data, just in different formats. Figure 8–4 illustrates the relationship between the Source, the Cursor objects, and the views.

Figure 8–4 A Source and Two Cursors for Different Views of the Values



9

Retrieving Query Results

This chapter describes how to retrieve the results of a query with an Oracle OLAP Java API `Cursor` and how to gain access to those results. This chapter also describes how to customize the behavior of a `Cursor` to fit your method of displaying the results. For information on the class hierarchies of `Cursor` and its related classes, and for information on the `Cursor` concepts of position, fetch size, and extent, see Chapter 8, "Understanding Cursor Classes and Concepts".

This chapter includes the following topics:

Retrieving the Results of a Query

A query is an OLAP Java API `Source` that specifies the data that you want to retrieve from the data store and any calculations that you want Oracle OLAP to perform on the data. A `Cursor` is the object that retrieves, or *fetches*, the result set specified by a `Source`. Creating a `Cursor` for a `Source` involves the following steps:

1. Get a primary `Source` from an `MdmObject` or create a derived `Source` through operations on a `DataProvider` or a `Source`. For information on getting or creating `Source` objects, see Chapter 5, "Understanding Source Objects".
2. If the `Source` is a derived `Source`, then commit the `Transaction` in which you created the `Source`. To commit the `Transaction`, call the `commitCurrentTransaction` method of your `TransactionProvider`. For more information on committing a `Transaction`, see Chapter 7, "Using a TransactionProvider". If the `Source` is a primary `Source`, then you do not need to commit the `Transaction`.
3. Create a `CursorManager` by calling a `createCursorManager` method of your `DataProvider` and passing that method the `Source`.
4. Create a `Cursor` by calling the `createCursor` method of the `CursorManager`.

Example 9–1 creates a `Cursor` for the derived `Source` named `querySource`. The example uses a `DataProvider` named `dp`. The example creates a `CursorManager` named `cursorMngr` and a `Cursor` named `queryCursor`.

Finally, the example closes the CursorManager. When you have finished using the Cursor, you should close the CursorManager to free resources.

Example 9–1 Creating a Cursor

```
CursorManager cursorMngr = dp.createCursorManager(querySource);
Cursor queryCursor = cursorMngr.createCursor();

// Use the Cursor in some way, such as to display its values.

cursorMngr.close();
```

Getting Values from a Cursor

The Cursor interface encapsulates the notion of a *current position* and has methods for moving the current position. The ValueCursor and CompoundCursor interfaces extend the Cursor interface. The Oracle OLAP Java API has implementations of the ValueCursor and CompoundCursor interfaces. Calling the createCursor method of a CursorManager returns either a ValueCursor or a CompoundCursor implementation, depending on the Source for which you are creating the Cursor.

A ValueCursor is returned for a Source that has a single set of values. A ValueCursor has a value at its current position, and it has methods for getting the value at the current position.

A CompoundCursor is created for a Source that has more than one set of values, which is a Source that has one or more outputs. Each set of values of the Source is represented by a child ValueCursor of the CompoundCursor. A CompoundCursor has methods for getting its child Cursor objects.

The structure of the Source determines the structure of the Cursor. A Source can have nested outputs, which occurs when one or more of the outputs of the Source is itself a Source with outputs. If a Source has a nested output, then the CompoundCursor for that Source has a child CompoundCursor for that nested output.

The CompoundCursor coordinates the positions of its child Cursor objects. The current position of the CompoundCursor specifies one set of positions of its child Cursor objects.

For an example of a Source that has only one level of output values, see Example 9–4. For an example of a Source that has nested output values, see Example 9–5.

An example of a Source that represents a single set of values is one returned by the getSource method of an MdmDimension, such as an MdmPrimaryDimension that represents product values. Creating a Cursor for that Source returns a ValueCursor. Calling the getCurrentValue method returns the product value at the current position of that ValueCursor.

Example 9–2 gets the Source from mdmProdHier, which is an MdmLevelHierarchy that represents product values, and creates a Cursor for that Source. The example sets the current position to the fifth element of the ValueCursor and gets the product value from the Cursor. The example then closes the CursorManager. In the example, dp is the DataProvider.

Example 9–2 Getting a Single Value from a ValueCursor

```
Source prodSource = mdmProdHier.getSource();
// Because prodSource is a primary Source, you do not need to
// commit the current Transaction.
```

```
CursorManager cursorMngr = dp.createCursorManager(prodSource);
Cursor prodCursor = cursorMngr.createCursor();
// Cast the Cursor to a ValueCursor.
ValueCursor prodValues = (ValueCursor) prodCursor;
// Set the position to the fifth element of the ValueCursor.
prodValues.setPosition(5);

// Product values are strings. Get the value at the current position.
String value = prodValues.getCurrentString();

// Do something with the value, such as display it.

// Close the CursorManager.
cursorMngr.close();
```

Example 9–3 uses the same `Cursor` as Example 9–2. Example 9–3 uses a `do...while` loop and the `next` method of the `ValueCursor` to move through the positions of the `ValueCursor`. The `next` method begins at a valid position and returns `true` when an additional position exists in the `Cursor`. It also advances the current position to that next position.

The example sets the position to the first position of the `ValueCursor`. The example loops through the positions and uses the `getCurrentValue` method to get the value at the current position.

Example 9–3 Getting All of the Values from a ValueCursor

```
// prodValues is the ValueCursor for prodSource.
prodValues.setPosition(1);
do
{
  println(prodValues.getCurrentValue);
} while(prodValues.next());
```

The values of the result set represented by a `CompoundCursor` are in the child `ValueCursor` objects of the `CompoundCursor`. To get those values, you must get the child `ValueCursor` objects from the `CompoundCursor`.

An example of a `CompoundCursor` is one that is returned by calling the `createCursor` method of a `CursorManager` for a `Source` that represents the values of a measure as specified by selected values from the dimensions of the measure.

Example 9–4 uses a `Source`, named `units`, that results from calling the `getSource` method of an `MdmMeasure` that represents the number of units sold. The dimensions of the measure are `MdmPrimaryDimension` objects representing products, customers, times, and channels. This example uses `Source` objects that represent selected values from the default hierarchies of those dimensions. The names of those `Source` objects are `prodSel`, `custSel`, `timeSel`, and `chanSel`. The creation of the `Source` objects representing the measure and the dimension selections is not shown.

Example 9–4 joins the dimension selections to the measure, which results in a `Source` named `unitsForSelections`. It creates a `CompoundCursor`, named `unitsForSelCursor`, for `unitsForSelections`, and gets the base `ValueCursor` and the outputs from the `CompoundCursor`. Each output is a `ValueCursor`, in this case. The outputs are returned in a `List`. The order of the outputs in the `List` is the inverse of the order in which the outputs were added to the list of outputs by the successive join operations. In the example, `dp` is the `DataProvider`.

Example 9–4 Getting ValueCursor Objects from a CompoundCursor

```
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
// Commit the current Transaction (code not shown).

// Create a Cursor for unitsForSelections.
CursorManager cursorMngr = dp.createCursorManager(unitsForSelections);
CompoundCursor unitsForSelCursor = (CompoundCursor)
                                    cursorMngr.createCursor();

// Get the base ValueCursor.
ValueCursor specifiedUnitsVals = unitsForSelCursor.getValueCursor();

// Get the outputs.
List outputs = unitsForSelCursor.getOutputs();
ValueCursor chanSelVals = (ValueCursor) outputs.get(0);
ValueCursor timeSelVals = (ValueCursor) outputs.get(1);
ValueCursor custSelVals = (ValueCursor) outputs.get(2);
ValueCursor prodSelVals = (ValueCursor) outputs.get(3);

// You can now get the values from the ValueCursor objects.
// When you have finished using the Cursor objects, close the CursorManager.
cursorMngr.close();
```

Example 9–5 uses the same units measure as Example 9–4, but it joins the dimension selections to the measure differently. Example 9–5 joins two of the dimension selections together. It then joins the result to the `Source` that results from joining the single dimension selections to the measure. The resulting `Source`, `unitsForSelections`, represents a query has nested outputs, which means it has more than one level of outputs.

The `CompoundCursor` that this example creates for `unitsForSelections` therefore also has nested outputs. The `CompoundCursor` has a child base `ValueCursor` and has as outputs three child `ValueCursor` objects and one child `CompoundCursor`.

Example 9–5 joins the selection of channel dimension values, `chanSel`, to the selection of customer dimension values, `custSel`. The result is `custByChanSel`, a `Source` that has customer values as the base values and channel values as the values of the output. The example joins to `units` the selections of product and time values, and then joins `custByChanSel`. The resulting query is represented by `unitsForSelections`.

The example commits the current `Transaction` and creates a `CompoundCursor`, named `unitsForSelCursor`, for `unitsForSelections`.

The example gets the base `ValueCursor` and the outputs from the `CompoundCursor`. In the example, `dp` is the `DataProvider`.

Example 9–5 Getting Values from a CompoundCursor with Nested Outputs

```
Source custByChanSel = custSel.join(chanSel);
Source unitsForSelections = units.join(prodSel)
                                 .join(timeSel)
                                 .join(custByChanSel);
// Commit the current Transaction (code not shown).

// Create a Cursor for unitsForSelections.
CursorManager cursorMngr = dp.createCursorManager(unitsForSelections);
```

```
Cursor unitsForSelCursor = cursorMngr.createCursor();

// Send the Cursor to a method that does different operations
// depending on whether the Cursor is a CompoundCursor or a
// ValueCursor.
printCursor(unitsForSelCursor);
cursorMngr.close();
// The remaining code of someMethod is not shown.

// The following code is in from the CursorPrintWriter class.
// The printCursor method has a do...while loop that moves through the positions
// of the Cursor passed to it. At each position, the method prints the number of
// the iteration through the loop and then a colon and a space. The output
// object is a PrintWriter. The method calls the private _printTuple method and
// then prints a new line. A "tuple" is the set of output ValueCursor values
// specified by one position of the parent CompoundCursor. The method prints one
// line for each position of the parent CompoundCursor.
private void printCursor(Cursor rootCursor)
{
  int i = 1;
  do
  {
     print(i++ + ": ");
     _printTuple(rootCursor);
     println();
     flush();
  } while(rootCursor.next());
}

// If the Cursor passed to the _printTuple method is a ValueCursor, then
// the method prints the value at the current position of the ValueCursor.
// If the Cursor passed in is a CompoundCursor, then the method gets the
// outputs of the CompoundCursor and iterates through the outputs,
// recursively calling itself for each output. The method then gets the
// base ValueCursor of the CompoundCursor and calls itself again.
private void _printTuple(Cursor cursor)
{
  if(cursor instanceof CompoundCursor)
  {
    CompoundCursor compoundCursor = (CompoundCursor)cursor;
    // Put an open parenthesis before the value of each output.
    print("(");
    Iterator iterOutputs = compoundCursor.getOutputs().iterator();
    Cursor output = (Cursor)iterOutputs.next();
    _printTuple(output);
    while(iterOutputs.hasNext())
    {
      // Put a comma after the value of each output.
      print(",");
      _printTuple((Cursor)iterOutputs.next());
    }
    // Put a comma after the value of the last output.
    print(",");
    // Get the base ValueCursor.
    _printTuple(compoundCursor.getValueCursor());

    // Put a close parenthesis after the base value to indicate
    // the end of the tuple.
    print(")");
  }
```

```
  else if(cursor instanceof ValueCursor)
  {
    ValueCursor valueCursor = (ValueCursor) cursor;
    if (valueCursor.hasCurrentValue())
      print(valueCursor.getCurrentValue());
    else                        // If this position has a null value.
      print("NA");
  }
}
```

Navigating a CompoundCursor for Different Displays of Data

With the methods of a `CompoundCursor` you can easily move through, or navigate, the `CompoundCursor` structure and get the values from the `ValueCursor` descendents of the `CompoundCursor`. Data from a multidimensional OLAP query is often displayed in a crosstab format, or as a table or a graph.

To display the data for multiple rows and columns, you loop through the positions at different levels of the `CompoundCursor` depending on the needs of your display. For some displays, such as a table, you loop through the positions of the parent `CompoundCursor`. For other displays, such as a crosstab, you loop through the positions of the child `Cursor` objects.

To display the results of a query in a table view, in which each row contains a value from each output `ValueCursor` and from the base `ValueCursor`, you determine the position of the top-level, or root, `CompoundCursor` and then iterate through its positions. Example 9–6 displays only a portion of the result set at one time. It creates a `Cursor` for a `Source` that represents a query that is based on a measure that has unit cost values. The dimensions of the measure are the product and time dimensions. The creation of the primary `Source` objects and the derived selections of the dimensions is not shown.

The example joins the `Source` objects representing the dimension value selections to the `Source` representing the measure. It commits the current `Transaction` and then creates a `Cursor`, casting it to a `CompoundCursor`. The example sets the position of the `CompoundCursor`, iterates through twelve positions of the `CompoundCursor`, and prints out the values specified at those positions. The `DataProvider` is `dp`.

Example 9–6   Navigating for a Table View

```
Source unitPriceByMonth = unitPrice.join(productSel)
                                   .join(timeSel);
// Commit the current Transaction (code not shown).

// Create a Cursor for unitPriceByMonth.
CursorManager cursorMngr = dp.createCursorManager(unitPriceByMonth);
CompoundCursor rootCursor = (CompoundCursor) cursorMngr.createCursor();

// Determine a starting position and the number of rows to display.
int start = 7;
int numRows =12;

println("Month     Product     Unit Price");
println("-------   --------   ----------");

// Iterate through the specified positions of the root CompoundCursor.
// Assume that the Cursor contains at least (start + numRows) positions.
for(int pos = start; pos < start + numRows; pos++)
{
```

```
     // Set the position of the root CompoundCursor.
     rootCursor.setPosition(pos);
    // Print the local values of the output and base ValueCursors.
    // The getLocalValue method gets the local value from the unique
    // value of a dimension element.
    String timeValue = ((ValueCursor)rootCursor.getOutputs().get(0))
                       .getCurrentString();
    String timeLocVal = getLocalValue(timeValue);
    String prodValue = ((ValueCursor)rootCursor.getOutputs().get(1))
                       .getCurrentString();
    String prodLocVal = getLocalValue(prodValue);
    Object price = rootCursor.getValueCursor().getCurrentValue();
    println(timeLocVal + "   " + prodLocVal + "   " +  price);
};
cursorMngr.close();
```

If the time selection for the query has eight values, such as the first month of each calendar quarter for the years 2001 and 2002, and the product selection has three values, then the result set of the `unitPriceByMonth` query has twenty-four positions. Example 9–6 displays the following table, which has the values specified by positions 7 through 18 of the `CompoundCursor`.

```
Month      Product    Unit Price
-------    --------   ----------
2001.07    ENVY ABM   2892.18
2001.07    ENVY EXE   3155.91
2001.07    ENVY STD   2505.57
2001.10    ENVY ABM   2856.86
2001.10    ENVY EXE   3105.53
2001.10    ENVY STD   2337.3
2002.01    ENVY ABM   2896.77
2002.01    ENVY EXE   3008.95
2002.01    ENVY STD   2140.71
2002.04    ENVY ABM   2880.39
2002.04    ENVY EXE   2953.96
2002.04    ENVY STD   2130.88
```

Example 9–7 uses the same query as Example 9–6. In a crosstab view, the first row is column headings, which are the values from `prodSel` in this example. The output for `prodSel` is the faster varying output because the `prodSel` dimension selection is the last output in the list of outputs that results from the operations that join the measure to the dimension selections. The remaining rows begin with a row heading. The row headings are values from the slower varying output, which is `timeSel`. The remaining positions of the rows, under the column headings, contain the `unitPrice` values specified by the set of the dimension values. To display the results of a query in a crosstab view, you iterate through the positions of the children of the top-level `CompoundCursor`.

The `DataProvider` is `dp`.

Example 9–7 Navigating for a Crosstab View Without Pages

```
Source unitPriceByMonth = unitPrice.join(productSel)
                                   .join(timeSel);
// Commit the current Transaction (code not shown).

// Create a Cursor for unitPriceByMonth.
CursorManager cursorMngr = dp.createCursorManager(unitPriceByMonth);
CompoundCursor rootCursor = (CompoundCursor) cursorMngr.createCursor();
```

```
// Get the outputs and the ValueCursor.
List outputs = rootCursor.getOutputs();
// The first output has the values of timeSel, the slower varying output.
ValueCursor rowCursor = (ValueCursor) outputs.get(0);
// The second output has the faster varying values of productSel.
ValueCursor columnCursor = (ValueCursor) outputs.get(1);
// The base ValueCursor has the values from unitPrice.
ValueCursor unitPriceValues = rootCursor.getValueCursor();

// Display the values as a crosstab.
println("                    PRODUCT");
println("          --------------------------------");
print("Month     ");
do
{
  String value = ((ValueCursor) columnCursor).getCurrentString();
  print(getContext().getLocalValue(value) + "    ");
} while (columnCursor.next());
println("\n-------   --------    --------    --------");

// Reset the column Cursor to its first element.
columnCursor.setPosition(1);

do
{
  // Print the row dimension values.
  String value = ((ValueCursor) rowCursor).getCurrentString();
  print(getContext().getLocalValue(value) + "    ");
  // Loop over columns.
  do
  {
    // Print data value.
    print(unitPriceValues.getCurrentValue() + "     ");
  } while (columnCursor.next());

  println();

  // Reset the column Cursor to its first element.
  columnCursor.setPosition(1);
  } while (rowCursor.next());

cursorMngr.close();
```

The following is a crosstab view of the values from the result set specified by the `unitPriceByMonth` query. The first line labels the rightmost three columns as having product values. The third line labels the first column as having month values and then labels each of the rightmost three columns with the product value for that column. The remaining lines have the month value in the left column and then have the data values from the units measure for the specified month and product.

```
                     PRODUCT
          --------------------------------
Month     ENVY ABM    ENVY EXE    ENVY STD
-------   --------    --------    --------
2001.01    3042.22     3223.28     2426.07
2001.04    3026.12     3107.65     2412.42
2001.07    2892.18     3155.91     2505.57
2001.10    2856.86     3105.53     2337.30
2002.01    2896.77     3008.95     2140.71
2002.04    2880.39     2953.96     2130.88
```

```
2002.07    2865.14     3002.34     2074.56
2002.10    2850.88     2943.96     1921.62
```

Example 9–8 creates a `Source` that is based on a measure of units sold values. The dimensions of the measure are the customer, product, time, and channel dimensions. The `Source` objects for the dimensions represent selections of the dimension values. The creation of those `Source` objects is not shown.

The query that results from joining the dimension selections to the measure `Source` represents unit sold values as specified by the values of the outputs.

The example creates a `Cursor` for the query and then sends the `Cursor` to the `printAsCrosstab` method, which prints the values from the `Cursor` in a crosstab. That method calls other methods that print page, column, and row values.

The fastest-varying output of the `Cursor` is the selection of products, which has three values (the product items ENVY ABM, ENVY EXE, and ENVY STD). The product values are the column headings of the crosstab. The next fastest-varying output is the selection of customers, which has three values (the customers COMP SERV TOK, COMP WHSE LON, and COMP WHSE SD). Those three values are the row headings. The page dimensions are selections of three time values (the months 2000.01, 2000.02, and 2000.03), and one channel value (DIR, which is the direct sales channel).

The `DataProvider` is `dp`. The `getLocalValue` method gets the local value from a unique dimension value.

Example 9–8   Navigating for a Crosstab View With Pages

```
// In someMethod.
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
// Commit the current Transaction (code not shown).

// Create a Cursor for unitsForSelections.
CursorManager cursorMngr = dp.createCursorManager(unitsForSelections);
CompoundCursor unitsForSelCursor = (CompoundCursor) cursorMngr.createCursor();

// Send the Cursor to the printAsCrosstab method.
printAsCrosstab(unitsForSelCursor);

cursorMngr.close();
// The remainder of the code of someMethod is not shown.

private void printAsCrosstab(CompoundCursor rootCursor)
{
  List outputs = rootCursor.getOutputs();
  int nOutputs = outputs.size();

  // Set the initial positions of all outputs.
  Iterator outputIter = outputs.iterator();
  while (outputIter.hasNext())
    ((Cursor) outputIter.next()).setPosition(1);

  // The last output is fastest-varying; it represents columns.
  // The next to last output represents rows.
  // All other outputs are on the page.
  Cursor colCursor = (Cursor) outputs.get(nOutputs - 1);
  Cursor rowCursor = (Cursor) outputs.get(nOutputs - 2);
  ArrayList pageCursors = new ArrayList();
```

```
  for (int i = 0 ; i < nOutputs - 2 ; i++)
  {
    pageCursors.add(outputs.get(i));
  }

  // Get the base ValueCursor, which has the data values.
  ValueCursor dataCursor = rootCursor.getValueCursor();

  // Print the pages of the crosstab.
  printPages(pageCursors, 0, rowCursor, colCursor, dataCursor);
}

// Prints the pages of a crosstab.
private void printPages(List pageCursors, int pageIndex, Cursor rowCursor,
                        Cursor colCursor, ValueCursor dataCursor)
{
  // Get a Cursor for this page.
  Cursor pageCursor = (Cursor) pageCursors.get(pageIndex);

  // Loop over the values of this page dimension.
  do
  {
    // If this is the fastest-varying page dimension, print a page.
    if (pageIndex == pageCursors.size() - 1)
    {
      // Print the values of the page dimensions.
      printPageHeadings(pageCursors);

      // Print the column headings.
      printColumnHeadings(colCursor);

      // Print the rows.
      printRows(rowCursor, colCursor, dataCursor);

      // Print a couple of blank lines to delimit pages.
      println();
      println();
    }

    // If this is not the fastest-varying page, recurse to the
    // next fastest-varying dimension.
    else
    {
      printPages(pageCursors, pageIndex + 1, rowCursor, colCursor,
                 dataCursor);
    }
  } while (pageCursor.next());

  // Reset this page dimension Cursor to its first element.
  pageCursor.setPosition(1);
}

// Prints the values of the page dimensions on each page.
private void printPageHeadings(List pageCursors)
{
  // Print the values of the page dimensions.
  Iterator pageIter = pageCursors.iterator();
  while (pageIter.hasNext())
  {
    String value = ((ValueCursor) pageIter.next()).getCurrentString();
```

```
    println(getLocalValue(value));
  }
  println();
}

// Prints the column headings on each page.
private void printColumnHeadings(Cursor colCursor)
{
  do
  {
     print("\t");
     String value = ((ValueCursor) colCursor).getCurrentString();
     print(getLocalValue(value));
  } while (colCursor.next());
  println();
  colCursor.setPosition(1);
}

// Prints the rows of each page.
private void printRows(Cursor rowCursor, Cursor colCursor,
                       ValueCursor dataCursor)
{
  // Loop over rows.
  do
  {
    // Print row dimension value.
    String value = ((ValueCursor) rowCursor).getCurrentString();
    print(getLocalValue(value));
    print("\t");
    // Loop over columns.
    do
    {
      // Print data value.
      print(dataCursor.getCurrentValue());
      print("\t");
    } while (colCursor.next());
    println();

    // Reset the column Cursor to its first element.
    colCursor.setPosition(1);
  } while (rowCursor.next());

  // Reset the row Cursor to its first element.
  rowCursor.setPosition(1);
}
```

Example 9–8 displays the following values, formatted as a crosstab. The display has added page, column, and row headings to identify the local values of the dimensions.

```
Channel DIR
Month 2001.01
                           Product
                ------------------------------
Customer        ENVY ABM   ENVY EXE   ENVY STD
-------------   --------   --------   --------
COMP WHSE SD       0          0          1
COMP SERV TOK      2          4          2
COMP WHSE LON      1          1          2
```

```
Channel DIR
Month 2000.02
                           Product
                ------------------------------
Customer        ENVY ABM   ENVY EXE   ENVY STD
-------------   --------   --------   --------
COMP WHSE SD       1          1          1
COMP SERV TOK      5          6          6
COMP WHSE LON      1          2          2


Channel DIR
Month 2000.03
                           Product
                ------------------------------
Customer        ENVY ABM   ENVY EXE   ENVY STD
-------------   --------   --------   --------
COMP WHSE SD       0          2          2
COMP SERV TOK      2          0          2
COMP WHSE LON      0          2          3
```

Specifying the Behavior of a Cursor

You can specify the following aspects of the behavior of a `Cursor`.

- The **fetch size** of a `Cursor`, which is the number of elements of the result set that the `Cursor` retrieves during one fetch operation.
- Whether or not Oracle OLAP calculates the **extent** of the `Cursor`. The extent is the total number of positions of the `Cursor`. The extent of a child `Cursor` of a `CompoundCursor` is relative to any of the slower varying outputs of the `CompoundCursor`.
- Whether or not Oracle OLAP calculates the positions in the parent `Cursor` at which the value of a child `Cursor` starts or ends.

To specify the behavior of `Cursor`, you use methods of a `CursorSpecification` that you specify for that `Cursor`. A `CursorSpecification` implements the `CursorInfoSpecification` interface.

You create a `CursorSpecification` for a `Source` by calling the `createCursorInfoSpecification` method of the `DataProvider`. You use methods of the `CursorSpecification` to set the characteristics that you want. You then create a `CursorManager` by calling the appropriate `createCursorManager` method of the `DataProvider`.

Note: Specifying the calculation of the extent or the starting or ending position in a parent `Cursor` of the current value of a child `Cursor` can be a very expensive operation. The calculation can require considerable time and computing resources. You should only specify these calculations when your application needs them.

For more information on the relationships of `Source`, `Cursor`, and `CursorSpecification` objects or the concepts of fetch size, extent, or `Cursor` positions, see Chapter 8.

Example 9–9 creates a Source, creates a CompoundCursorSpecification for a Source, and then gets the child CursorSpecification objects from the top-level CompoundCursorSpecification.

Example 9–9 Getting CursorSpecification Objects for a Source

```
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
// Commit the current Transaction (code not shown).

// Create a CompoundCursorSpecification for unitsForSelections.
CompoundCursorSpecification rootCursorSpec = (CompoundCursorSpecification)
                      dp.createCursorInfoSpecification(unitsForSelections);

// Get the ValueCursorSpecification for the base values.
ValueCursorSpecification baseValueSpec =
                        rootCursorSpec.getValueCursorSpecification();

// Get the ValueCursorSpecification objects for the outputs.
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification chanSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(0);
ValueCursorSpecification timeSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(1);
ValueCursorSpecification prodSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(2);
ValueCursorSpecification custSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(3);
```

Once you have the CursorSpecification objects, you can use their methods to specify the behavior of the Cursor objects that correspond to them.

Calculating Extent and Starting and Ending Positions of a Value

To manage the display of the result set retrieved by a CompoundCursor, you sometimes need to know the extent of the child Cursor components. You might also want to know the position at which the current value of a child Cursor starts in the parent CompoundCursor. You might want to know the **span** of the current value of a child Cursor. The span is the number of positions of the parent Cursor that the current value of the child Cursor occupies. You can calculate the span by subtracting the starting position of the value from the ending position and subtracting 1.

Before you can get the extent of a Cursor or get the starting or ending positions of a value in the parent Cursor, you must specify that you want Oracle OLAP to calculate the extent or those positions. To specify the performance of those calculations, you use methods of the CursorSpecification for the Cursor.

Example 9–10 specifies calculating the extent of a Cursor. The example uses the CompoundCursorSpecification from Example 9–9.

Example 9–10 Specifying the Calculation of the Extent of a Cursor

```
rootCursorSpec.setExtentCalculationSpecified(true);
```

You can use methods of a `CursorSpecification` to determine whether the `CursorSpecification` specifies the calculation of the extent of a `Cursor` as in the following example.

```
boolean isSet = rootCursorSpec.isExtentCalculationSpecified();
```

Example 9–11 specifies calculating the starting and ending positions of the current value of a child `Cursor` in the parent `Cursor`. The example uses the `CompoundCursorSpecification` from Example 9–9.

Example 9–11 Specifying the Calculation of Starting and Ending Positions in a Parent

```
// Get the List of CursorSpecification objects for the outputs.
// Iterate through the list, specifying the calculation of the extent
// for each output CursorSpecification.
Iterator iterOutputSpecs = rootCursorSpec.getOutputs().iterator();
while(iterOutputSpecs.hasNext())
{
  ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)
                                           iterOutputSpecs.next();
  valCursorSpec.setParentStartCalculationSpecified(true);
  valCursorSpec.setParentEndCalculationSpecified(true);
}
```

You can use methods of a `CursorSpecification` to determine whether the `CursorSpecification` specifies the calculation of the starting or ending positions of the current value of a child `Cursor` in its parent `Cursor`, as in the following example.

```
Iterator iterOutputSpecs = rootCursorSpec.getOutputs().iterator();
ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)
                                         iterOutputSpecs.next();
while(iterOutputSpecs.hasNext())
{
  if (valCursorSpec.isParentStartCalculationSpecified())
    // Do something.
  if (valCursorSpec.isParentEndCalculationSpecified())
    // Do something.
  valCursorSpec = (ValueCursorSpecification) iterOutputSpecs.next();
}
```

Example 9–12 determines the span of the positions in a parent `CompoundCursor` of the current value of a child `Cursor` for two of the outputs of the `CompoundCursor`. The example uses the `unitForSelections` `Source` from Example 9–8.

The example gets the starting and ending positions of the current values of the time and product selections and then calculates the span of those values in the parent `Cursor`. The parent is the root `CompoundCursor`. The `DataProvider` is `dp`.

Example 9–12 Calculating the Span of the Positions in the Parent of a Value

```
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
// Commit the current Transaction (code not shown).

// Create a CompoundCursorSpecification for unitsForSelections.
CompoundCursorSpecification rootCursorSpec = (CompoundCursorSpecification)
                          dp.createCursorInfoSpecification(unitsForSelections);
```

```
// Get the CursorSpecification objects for the outputs.
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification timeSelValCSpec =
  (ValueCursorSpecification) outputSpecs.get(1); // Output for time.
ValueCursorSpecification prodSelValCSpec =
 (ValueCursorSpecification) outputSpecs.get(3);  // Output for product.

// Specify the calculation of the starting and ending positions.
timeSelValCSpec.setParentStartCalculationSpecified(true);
timeSelValCSpec.setParentEndCalculationSpecified(true);
prodSelValCSpec.setParentStartCalculationSpecified(true);
prodSelValCSpec.setParentEndCalculationSpecified(true);

// Create the CursorManager and the Cursor.
CursorManager cursorMngr = dp.createCursorManager(unitsForSelections,
                                                  100, rootCursorSpec);
CompoundCursor rootCursor = (CompoundCursor) cursorMngr.createCursor();

// Get the child Cursor objects.
ValueCursor baseValCursor = cursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor chanSelVals = (ValueCursor) outputs.get(0);
ValueCursor timeSelVals = (ValueCursor) outputs.get(1);
ValueCursor custSelVals = (ValueCursor) outputs.get(2);
ValueCursor prodSelVals = (ValueCursor) outputs.get(3);

// Set the position of the root CompoundCursor.
rootCursor.setPosition(15);

// Get the values at the current position and determine the span
// of the values of the time and product outputs.
print(promoSelVals.getCurrentValue() + ", ");
print(chanSelVals.getCurrentValue() + ", ");
print(timeSelVals.getCurrentValue() + ", ");
print(custSelVals.getCurrentValue() + ", ");
print(prodSelVals.getCurrentValue() + ", ");
println(baseValCursor.getCurrentValue());

// Determine the span of the values of the two fastest-varying outputs.
int span;
span = (prodSelVals.getParentEnd() - prodSelVals.getParentStart()) +1);
println("The span of " + prodSelVals.getCurrentValue() +
        " at the current position is " + span + ".")
span = (timeSelVals.getParentEnd() - timeSelVals.getParentStart()) +1);
println("The span of " + timeSelVals.getCurrentValue() +
        " at the current position is " + span + ".")
cursorMngr.close();
```

This example displays the following text.

```
CHANNEL_PRIMARY::CHANNEL::DIR, CALENDAR_YEAR::MONTH::2000.02,
SHIPMENTS::SHIP_TO::COMP SERV TOK, PRODUCT_PRIMARY::ITEM::ENVY STD, 6.0
The span of PRODUCT_PRIMARY::ITEM::ENVY STD at the current position is 1.
The span of CALENDAR_YEAR::MONTH::2000.02 at the current position is 9.
```

Specifying a Fetch Size

The number of elements of a `Cursor` that Oracle OLAP sends to the client application during one fetch operation depends on the fetch size specified for that `Cursor`. The default fetch size is 100. To change the fetch size, you can set the fetch size on the root `Cursor` for a `Source`.

Example 9–13 gets the default fetch size from the `CompoundCursorSpecification` from Example 9–9. The example creates a `Cursor` and sets a different fetch size on it, and then gets the fetch size for the `Cursor`. The `DataProvider` is `dp`.

Example 9–13 Specifying a Fetch Size

```
println("The default fetch size is "
         + rootCursorSpec.getDefaultFetchSize() + ".");
Source source = rootCursorSpec.getSource();
CursorManager cursorMngr = dp.createCursorManager(source);
Cursor rootCursor = cursorMngr.createCursor();
rootCursor.setFetchSize(10);
println("The fetch size is now " + rootCursor.getFetchSize()) + ".";
```

This example displays the following text.

```
The default fetch size is 100.
The fetch size is now 10.
```

10

Creating Dynamic Queries

This chapter describes the Oracle OLAP Java API `Template` class and the related classes that you use to create dynamic queries. This chapter also provides examples of implementations of those classes.

This chapter includes the following topics:

- About Template Objects
- Overview of Template and Related Classes
- Designing and Implementing a Template

About Template Objects

The `Template` class is the basis of a very powerful feature of the Oracle OLAP Java API. You use `Template` objects to create modifiable `Source` objects. With those `Source` objects, you can create dynamic queries that can change in response to end-user selections. `Template` objects also offer a convenient way for you to translate user-interface elements into OLAP Java API operations and objects.

These features are briefly described in the following section. The rest of this chapter describes the `Template` class and the other classes you use to create dynamic `Source` objects. For information on the `Transaction` objects that you use to make changes to the dynamic `Source` and to either save or discard those changes, see Chapter 7, "Using a TransactionProvider".

About Creating a Dynamic Source

The main feature of a `Template` is its ability to produce a dynamic `Source`. That ability is based on two of the other objects that a `Template` uses: instances of the `DynamicDefinition` and `MetadataState` classes.

When a `Source` is created, Oracle OLAP automatically associates a `SourceDefinition` with it. The `SourceDefinition` has information about the `Source`. Once created, the `Source` and the associated `SourceDefinition` are associated immutably. The `getSource` method of a `SourceDefinition` returns the `Source` associated with it.

`DynamicDefinition` is a subclass of `SourceDefinition`. A `Template` creates a `DynamicDefinition`, which acts as a proxy for the `SourceDefinition` of the `Source` produced by the `Template`. This means that instead of always getting the same immutably associated `Source`, the `getSource` method of the `DynamicDefinition` gets whatever `Source` is currently produced by the `Template`. The instance of the `DynamicDefinition` does not change even though the `Source` that it gets is different.

The Source that a Template produces can change because the values, including other Source objects, that the Template uses to create the Source can change. A Template stores those values in a MetadataState. A Template provides methods to get the current state of the MetadataState, to get or set a value, and to set the state. You use those methods to change the data values the MetadataState stores.

You use a DynamicDefinition to get the Source produced by a Template. If your application changes the state of the values that the Template uses to create the Source, for example, in response to end-user selections, then the application uses the same DynamicDefinition to get the Source again, even though the new Source defines a result set different than the previous Source.

The Source produced by a Template can be the result of a series of Source operations that create other Source objects, such as a series of selections, sorts, calculations, and joins. You put the code for those operations in the generateSource method of a SourceGenerator for the Template. That method returns the Source produced by the Template. The operations use the data stored in the MetadataState.

You might build an extremely complex query that involves the interactions of dynamic Source objects produced by many different Template objects. The end result of the query building is a Source that defines the entire complex query. If you change the state of any one of the Template objects that you used to create the final Source, then the final Source represents a result set that is different from that of the previous Source. You can thereby modify the final query without having to reproduce all of the operations involved in defining the query.

About Translating User Interface Elements into OLAP Java API Objects

You design Template objects to represent elements of the user interface of an application. Your Template objects turn the selections that the end user makes into OLAP Java API query-building operations that produce a Source. You then create a Cursor to fetch the result set defined by the Source from Oracle OLAP. You get the values from the Cursor and display them to the end user. When an end user makes changes to the selections, you change the state of the Template. You then get the Source produced by the Template, create a new Cursor, get the new values, and display them.

Overview of Template and Related Classes

In the OLAP Java API, several classes work together to produce a dynamic Source. In designing a Template, you must implement or extend the following:

- The Template abstract class
- The MetadataState interface
- The SourceGenerator interface

Instances of those three classes, plus instances of the DataProvider and DynamicDefinition classes, work together to produce the Source that the Template defines.

What Is the Relationship Between the Classes That Produce a Dynamic Source?

The classes that produce a dynamic Source work together as follows:

- A Template has methods that create a DynamicDefinition and that get and set the current state of a MetadataState. An extension to the Template abstract class adds methods that get and set the values of fields on the MetadataState.
- The MetadataState implementation has fields for storing the data to use in generating the Source for the Template. When you create a new Template, you pass the MetadataState to the constructor of the Template. When you call the getSource method of the DynamicDefinition, the MetadataState is passed to the generateSource method of the SourceGenerator.
- The DataProvider is used in creating a Template and by the SourceGenerator in creating new Source objects.
- The SourceGenerator implementation has a generateSource method that uses the current state of the data in the MetadataState to produce a Source for the Template. You pass in the SourceGenerator to the createDynamicDefinition method of the Template to create a DynamicDefinition.
- The DynamicDefinition has a getSource method that gets the Source produced by the SourceGenerator. The DynamicDefinition serves as a proxy for the SourceDefinition that is immutably associated with the Source.

Template Class

You use a Template to produce a modifiable Source. A Template has methods for creating a DynamicDefinition and for getting and setting the current state of the Template. In extending the Template class, you add methods that provide access to the fields on the MetadataState for the Template. The Template creates a DynamicDefinition that you use to get the Source produced by the SourceGenerator for the Template.

For an example of a Template implementation, see Example 10–1.

MetadataState Interface

An implementation of the MetadataState interface stores the current state of the values for a Template. A MetadataState must include a clone method that creates a copy of the current state.

When instantiating a new Template, you pass a MetadataState to the Template constructor. The Template has methods for getting and setting the values stored by the MetadataState. The generateSource method of the SourceGenerator for the Template uses the MetadataState when the method produces a Source for the Template.

For an example of a MetadataState implementation, see Example 10–2.

SourceGenerator Interface

An implementation of SourceGenerator must include a generateSource method, which produces a Source for a Template. A SourceGenerator must produce only one type of Source, such as a BooleanSource, a NumberSource, or a StringSource. In producing the Source, the generateSource method uses the current state of the data represented by the MetadataState for the Template.

To get the Source produced by the generateSource method, you create a DynamicDefinition by passing the SourceGenerator to the createDynamicDefinition method of the Template. You then get the Source by calling the getSource method of the DynamicDefinition.

A Template can create more than one DynamicDefinition, each with a differently implemented SourceGenerator. The generateSource methods of the different SourceGenerator objects use the same data, as defined by the current state of the MetadataState for the Template, to produce Source objects that define different queries.

For an example of a SourceGenerator implementation, see Example 10–3.

DynamicDefinition Class

DynamicDefinition is a subclass of SourceDefinition. You create a DynamicDefinition by calling the createDynamicDefinition method of a Template and passing it a SourceGenerator. You get the Source produced by the SourceGenerator by calling the getSource method of the DynamicDefinition.

A DynamicDefinition created by a Template is a proxy for the SourceDefinition of the Source produced by the SourceGenerator. The SourceDefinition is immutably associated with the Source. If the state of the Template changes, then the Source produced by the SourceGenerator is different. Because the DynamicDefinition is a proxy, you use the same DynamicDefinition to get the new Source even though that Source has a different SourceDefinition.

The getCurrent method of a DynamicDefinition returns the SourceDefinition immutably associated with the Source that the generateSource method currently returns. For an example of the use of a DynamicDefinition, see Example 10–4.

Designing and Implementing a Template

The design of a Template reflects the query-building elements of the user interface of an application. For example, suppose you want to develop an application that allows the end user to create a query that requests a number of values from the top or bottom of a list of values. The values are from one dimension of a measure. The other dimensions of the measure are limited to single values.

The user interface of your application has a dialog box that allows the end user to do the following:

- Select a radio button that specifies whether the data values should be from the top or bottom of the range of values.
- Select a measure from a drop-down list of measures.
- Select a number from a field. The number specifies the number of data values to display.
- Select one of the dimensions of the measure as the base of the data values to display. For example, if the user selects the product dimension, then the query specifies some number of products from the top or bottom of the list of products. The list is determined by the measure and the selected values of the other dimensions.
- Click a button to bring up a dialog box through which the end user selects the single values for the other dimensions of the selected measure. After selecting the

values of the dimensions, the end user clicks an OK button on the second dialog box and returns to the first dialog box.

- Click an OK button to generate the query. The results of the query appear.

To generate a `Source` that represents the query that the end user creates in the first dialog box, you design a `Template` called `TopBottomTemplate`. You also design a second `Template`, called `SingleSelectionTemplate`, to create a `Source` that represents the end user's selections of single values for the dimensions other than the base dimension. The designs of your `Template` objects reflect the user interface elements of the dialog boxes.

In designing the `TopBottomTemplate` and its `MetadataState` and `SourceGenerator`, you do the following:

- Create a class called `TopBottomTemplate` that extends `Template`. To the class, you add methods that get the current state of the `Template`, set the values specified by the user, and then set the current state of the `Template`.
- Create a class called `TopBottomTemplateState` that implements `MetadataState`. You provide fields on the class to store values for the `SourceGenerator` to use in generating the `Source` produced by the `Template`. The values are set by methods of the `TopBottomTemplate`.
- Create a class called `TopBottomTemplateGenerator` that implements `SourceGenerator`. In the `generateSource` method of the class, you provide the operations that create the `Source` specified by the end user's selections.

Using your application, an end user selects units sold as the measure and products as the base dimension in the first dialog box. The end user also selects the Asia Pacific region, the first quarter of 2001, and the direct sales channel as the single values for each of the remaining dimensions.

The query that the end user has created requests the ten products that have the highest total amount of units sold through the direct sales channel to customers in the Asia Pacific region during the calendar year 2001.

For examples of implementations of the `TopBottomTemplate`, `TopBottomTemplateState`, and `TopBottomTemplateGenerator` classes, and an example of an application that uses them, see Example 10–1, Example 10–2, Example 10–3, and Example 10–4. The `TopBottomTemplateState` and `TopBottomTemplateGenerator` classes are implemented as inner classes of the `TopBottomTemplate` outer class.

Implementing the Classes for a Template

Example 10–1 is an implementation of the `TopBottomTemplate` class.

Example 10–1 Implementing a Template

```
import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.DynamicDefinition;
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.SourceGenerator;
import oracle.olapi.data.source.Template;
import oracle.olapi.transaction.metadataStateManager.MetadataState;

/**
 * Creates a TopBottomTemplateState, a TopBottomTemplateGenerator,
 * and a DynamicDefinition.
 * Gets the current state of the TopBottomTemplateState and the values
```

```
 * that it stores.
 * Sets the data values stored by the TopBottomTemplateState and sets the
 * changed state as the current state.
 */
public class TopBottomTemplate extends Template
{
// Constants for specifying the selection of elements from the
// beginning or the end of the result set.
  public static final int TOP_BOTTOM_TYPE_TOP = 0;
  public static final int TOP_BOTTOM_TYPE_BOTTOM = 1;

  // Variable to store the DynamicDefinition.
  private DynamicDefinition dynamicDef;

  /**
   * Creates a TopBottomTemplate with a default type and number values
   * and the specified base dimension.
   */
  public TopBottomTemplate(Source base, DataProvider dataProvider)
  {
    super(new TopBottomTemplateState(base, TOP_BOTTOM_TYPE_TOP, 0),
                                     dataProvider);
    // Create the DynamicDefinition for this Template. Create the
    // TopBottomTemplateGenerator that the DynamicDefinition uses.
    dynamicDef =
    createDynamicDefinition(new TopBottomTemplateGenerator(dataProvider));
  }

  /**
   * Gets the Source produced by the TopBottomTemplateGenerator
   * from the DynamicDefinition.
   */
  public final Source getSource()
  {
    return dynamicDef.getSource();
  }

  /**
   * Gets the Source that is the base of the elements in the result set.
   * Returns null if the state has no base.
   */
  public Source getBase()
  {
    TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    return state.base;
  }

  /**
   * Sets a Source as the base.
   */
  public void setBase(Source base)
  {
     TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
    state.base = base;
    setCurrentState(state);
  }

  /**
   * Gets the Source that specifies the measure and the single
   * selections from the dimensions other than the base.
```

```
 */
public Source getCriterion()
{
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  return state.criterion;
}

/**
 * Specifies a Source that defines the measure and the single values
 * selected from the dimensions other than the base.
 * The SingleSelectionTemplate produces such a Source.
 */
public void setCriterion(Source criterion)
{
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  state.criterion = criterion;
  setCurrentState(state);
}

/**
 * Gets the type, which is either TOP_BOTTOM_TYPE_TOP or
 * TOP_BOTTOM_TYPE_BOTTOM.
 */
public int getTopBottomType()
{
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  return state.topBottomType;
}

/**
 * Sets the type.
 */
public void setTopBottomType(int topBottomType)
{
  if ((topBottomType < TOP_BOTTOM_TYPE_TOP) ||
      (topBottomType > TOP_BOTTOM_TYPE_BOTTOM))
    throw new IllegalArgumentException("InvalidTopBottomType");
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  state.topBottomType = topBottomType;
  setCurrentState(state);
}

/**
 * Gets the number of values selected.
 */
public float getN()
{
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  return state.N;
}

/**
 * Sets the number of values to select.
 */
public void setN(float N)
{
  TopBottomTemplateState state = (TopBottomTemplateState) getCurrentState();
  state.N = N;
  setCurrentState(state);
}
```

```
}
```

Example 10–2 is an implementation of the `TopBottomTemplateState` inner class.

Example 10–2 Implementing a MetadataState

```
/**
 * Stores data that can be changed by its TopBottomTemplate.
 * The data is used by a TopBottomTemplateGenerator in producing
 * a Source for the TopBottomTemplate.
 */
private static final class TopBottomTemplateState
     implements Cloneable, MetadataState
{
  public int topBottomType;
  public float N;
  public Source criterion;
  public Source base;

  /**
   * Creates a TopBottomTemplateState.
   */
  public TopBottomTemplateState(Source base, int topBottomType, float N)
  {
    this.base = base;
    this.topBottomType = topBottomType;
    this.N = N;
  }

  /**
   * Creates a copy of this TopBottomTemplateState.
   */
  public final Object clone()
  {
    try
    {
      return super.clone();
    }
    catch(CloneNotSupportedException e)
    {
      return null;
    }
  }
}
```

Example 10–3 is an implementation of the `TopBottomTemplateGenerator` inner class.

Example 10–3 Implementing a SourceGenerator

```
/**
 * Produces a Source for a TopBottomTemplate based on the data
 * values of a TopBottomTemplateState.
 */
private final class TopBottomTemplateGenerator
      implements SourceGenerator
{
  // Store the DataProvider.
  private DataProvider _dataProvider;
```

```
  /**
   * Creates a TopBottomTemplateGenerator.
   */
  public TopBottomTemplateGenerator(DataProvider dataProvider)
  {
    _dataProvider = dataProvider;
  }

  /**
   * Generates a Source for a TopBottomTemplate using the current
   * state of the data values stored by the TopBottomTemplateState.
   */
  public Source generateSource(MetadataState state)
  {
    TopBottomTemplateState castState = (TopBottomTemplateState) state;
    if (castState.criterion == null)
      throw new NullPointerException("CriterionParameterMissing");
    Source sortedBase = null;

    // Depending on the topBottomType value, select from the base Source
    // the elements specified by the criterion Source and sort the
    // elements in ascending or descending order.
    // For descending order, specify that null values are last.
    // For ascending order, specify that null values are first.

    if (castState.topBottomType == TOP_BOTTOM_TYPE_TOP)
      sortedBase = castState.base.sortDescending(castState.criterion, false);
    else
      sortedBase = castState.base.sortAscending(castState.criterion, true);
    return sortedBase.interval(1, Math.round(castState.N));
  }
}
```

Implementing an Application That Uses Templates

After you have stored the selections made by the end user in the `MetadataState` for the `Template`, use the `getSource` method of the `DynamicDefinition` to get the dynamic `Source` created by the `Template`. This section provides an example of an application that uses the `TopBottomTemplate` described in Example 10–1. For brevity, the code does not contain much exception handling.

The `BaseExample11g` class creates and stores an instance of the `Context11g` class, which has methods that do the following:

- Connects to an Oracle Database instance as the user in the command line arguments.
- Creates `Cursor` objects and displays their values.

Example 10–4 does the following:

- Gets the `MdmMetadataProvider` and the `MdmRootSchema`.
- Gets the `DataProvider`.
- Gets the `MdmDatabaseSchema` for the user.
- Gets the `MdmCube` that has the `UNITS` and `SALES` measures. From the cube, the example gets the measures and the dimensions.

- Creates a `SingleSelectionTemplate` for selecting single values from some of the dimensions of the measure. For the code of the `SingleSelectionTemplate` class that this example uses, see Appendix B.
- Creates a `TopBottomTemplate` and stores selections made by the end user.
- Gets the `Source` produced by the `TopBottomTemplate`.
- Uses the `Context11g` object to create a `Cursor` for that `Source` and to display its values.

To use Example 7–3, replace the lines in the `run` method from the following comment to the end of the method:

```
// Replace from here on for the Using Child Transaction example.
```

Example 10–4 Getting the Source Produced by the Template

```
import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.Source;
import oracle.olapi.examples.*;
import oracle.olapi.metadata.mdm.MdmAttribute;
import oracle.olapi.metadata.mdm.MdmBaseMeasure;
import oracle.olapi.metadata.mdm.MdmCube;
import oracle.olapi.metadata.mdm.MdmDatabaseSchema;
import oracle.olapi.metadata.mdm.MdmDimensionLevel;
import oracle.olapi.metadata.mdm.MdmDimensionMemberInfo;
import oracle.olapi.metadata.mdm.MdmHierarchyLevel;
import oracle.olapi.metadata.mdm.MdmLevelHierarchy;
import oracle.olapi.metadata.mdm.MdmMetadataProvider;
import oracle.olapi.metadata.mdm.MdmPrimaryDimension;
import oracle.olapi.metadata.mdm.MdmRootSchema;

/**
 * Creates a query that specifies a number of elements from the top
 * or bottom of a selection of dimension members, creates a Cursor
 * for the query, and displays the values of the Cursor.
 * The selected dimension members are those that have measure values
 * that are specified by selected members of the other dimensions of
 * the measure.
 */
public class TopBottomTest extends BaseExample11g
{
  /**
   * Gets the UNITS_CUBE_AWJ MdmCube.
   * From the cube, gets the MdmPrimaryDimension objects and the
   * MdmBaseMeasure objects for the UNITS and SALES measures.
   * Gets the default hierarchies for the dimensions and then gets the Source
   * object for the base of the query.
   * Creates a SingleSelectionTemplate and adds selections to it.
   * Creates a TopBottomTemplate and sets its properties.
   * Gets the Source produced by the TopBottomTemplate, creates a Cursor
   * for it, and displays the values of the Cursor.
   * Changes the state of the SingleSelectionTemplate and the
   * TopBottomTemplate, creates a new Cursor for the Source produced by the
   * TopBottomTemplate, and displays the values of that Cursor.
   */
  public void run() throws Exception
  {
    // Get the MdmMetadataProvider from the superclass.
    MdmMetadataProvider metadataProvider = getMdmMetadataProvider();
    // Get the DataProvider from the Context11g object of the superclass.
```

```
DataProvider dp = getContext().getDataProvider();

// Get the MdmRootSchema and the MdmDatabaseSchema for the user.
MdmRootSchema mdmRootSchema =
                           (MdmRootSchema)metadataProvider.getRootSchema();
MdmDatabaseSchema mdmDBSchema =
                   mdmRootSchema.getDatabaseSchema(getContext().getUser());

MdmCube unitsCube =
       (MdmCube)mdmDBSchema.getTopLevelObject("UNITS_CUBE_AWJ");
MdmBaseMeasure mdmUnits = (MdmBaseMeasure)unitsCube.getMeasure("UNITS");
MdmBaseMeasure mdmSales = (MdmBaseMeasure)unitsCube.getMeasure("SALES");


// Get the Source objects for the measures.
Source units = mdmUnits.getSource();
Source sales = mdmSales.getSource();

// Get the MdmPrimaryDimension objects for the dimensions of the cube.
List<MdmPrimaryDimension> cubeDims = unitsCube.getDimensions();
MdmPrimaryDimension mdmTimeDim = null;
MdmPrimaryDimension mdmProdDim = null;
MdmPrimaryDimension mdmCustDim = null;
MdmPrimaryDimension mdmChanDim = null;

for(MdmPrimaryDimension mdmPrimDim : cubeDims)
{
  if (mdmPrimDim.getName().startsWith("TIME"))
    mdmTimeDim = mdmPrimDim;
  else if (mdmPrimDim.getName().startsWith("PROD"))
    mdmProdDim = mdmPrimDim;
  else if (mdmPrimDim.getName().startsWith("CUST"))
    mdmCustDim = mdmPrimDim;
  else if (mdmPrimDim.getName().startsWith("CHAN"))
    mdmChanDim = mdmPrimDim;
}

// Get the default hierarchy of the Product dimension.
MdmLevelHierarchy mdmProdHier = (MdmLevelHierarchy)
                                 mdmProdDim.getDefaultHierarchy();

// Get the detail dimenson level of the Product dimension.
MdmDimensionLevel mdmItemDimLevel =
                            mdmProdDim.findOrCreateDimensionLevel("ITEM");
// Get the default hierarchy of the Product dimension.
MdmLevelHierarchy mdmProdHier = (MdmLevelHierarchy)
                                mdmProdDim.getDefaultHierarchy();
// Get the hierarchy level of the dimension level.
MdmHierarchyLevel mdmItemHierLevel =
                 mdmProdHier.findOrCreateHierarchyLevel(mdmItemDimLevel);
// Get the Source for the hierarchy level.
Source itemLevel = mdmItemHierLevel.getSource();

// Get the short description attribute for the Product dimension and
// the Source for the attribute.
MdmAttribute mdmProdShortDescrAttr =
                       mdmProdDim.getShortValueDescriptionAttribute();
Source prodShortDescrAttr = mdmProdShortDescrAttr.getSource();

// Create a SingleSelectionTemplate to produce a Source that
```

```
// represents the measure values specified by single members of each of
// the dimensions of the measure other than the base dimension.
SingleSelectionTemplate singleSelections =
                    new SingleSelectionTemplate(units, dp);

// Create MdmDimensionMemberInfo objects for single members of the
// other dimensions of the measure.
MdmDimensionMemberInfo timeMemInfo =
     new MdmDimensionMemberInfo(mdmTimeDim,
                               "CALENDAR_YEAR::YEAR::CY2001");
MdmDimensionMemberInfo custMemInfo =
     new MdmDimensionMemberInfo(mdmCustDim,
                               "SHIPMENTS::REGION::APAC");
MdmDimensionMemberInfo chanMemInfo =
     new MdmDimensionMemberInfo(mdmChanDim,
                               "CHANNEL_PRIMARY::CHANNEL::DIR");

// Add the dimension member information objects to the
// SingleSelectionTemplate.
singleSelections.addDimMemberInfo(custMemInfo);
singleSelections.addDimMemberInfo(chanMemInfo);
singleSelections.addDimMemberInfo(timeMemInfo);

// Create a TopBottomTemplate specifying, as the base, the Source for a
// a level of a hierarchy.
TopBottomTemplate topNBottom = new TopBottomTemplate(itemLevel, dp);

// Specify whether to retrieve the elements from the beginning (top) or the
// end (bottom) of the selected elements of the base dimension.
topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);

// Set the number of elements of the base dimension to retrieve.
topNBottom.setN(10);
// Get the Source produced by the SingleSelectionTemplate and specify it as
// the criterion object.
topNBottom.setCriterion(singleSelections.getSource());

// Replace from here on for the Using Child Transaction Objects example.

// Display a description of the result.
String resultDescription = " products with the most units sold \nfor";
displayResultDescr(singleSelections, topNBottom, resultDescription);

// Get the Source produced by the TopBottomTemplate.
Source result = topNBottom.getSource();

// Join the Source produced by the TopBottomTemplate with the short
// value descriptions. Use the joinHidden method so that the
// dimension member values do not appear in the result.
Source result = prodShortDescrAttr.joinHidden(topNBottomResult);

// Commit the current transaction.
getContext().commit();  // Method of Context11g.

// Create a Cursor for the result and display the values of the Cursor.
getContext().displayTopBottomResult(result);

// Change a dimension member selection of the SingleSelectionTemplate.
timeMemInfo.setUniqueValue("CALENDAR_YEAR::YEAR::CY2000");
singleSelections.changeSelection(timeMemInfo);
```

```
  // Change the number of elements selected and the type of selection.
  topNBottom.setN(5);
  topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);

  // Join the Source produced by the TopBottomTemplate to the short
  // description attribute.
  result = prodShortDescrAttr.joinHidden(topNBottomResult);

  // Commit the current transaction.
  getContext().commit();

  // Display a description of the result.
  resultDescription = " products with the fewest units sold \nfor";
  displayResultDescr(singleSelections, topNBottom, resultDescription);

  // Create a new Cursor for the Source produced by the TopBottomTemplate
  // and display the Cursor values.
  getContext().displayTopBottomResult(result);

  // Now change the measure to Sales, and get the top and bottom products by
  // Sales.
  singleSelections.setMeasure(sales);
  // Change the number of elements selected.
  topNBottom.setN(7);
  // Change the type of selection back to the top.
  topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);

  resultDescription = " products with the highest sales amounts \nfor";
  displayResultDescr(singleSelections, topNBottom, resultDescription);

  topNBottomResult = topNBottom.getSource();
  result = prodShortDescrAttr.joinHidden(topNBottomResult);

  // Commit the current transaction.
  getContext().commit();
  getContext().displayTopBottomResult(result);

  // Change the type of selection back to the bottom.
  topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);

  resultDescription = " products with the lowest sales amounts \nfor";
  displayResultDescr(singleSelections, topNBottom, resultDescription);

  topNBottomResult = topNBottom.getSource();
  result = prodShortDescrAttr.joinHidden(topNBottomResult);

  // Commit the current transaction.
  getContext().commit();
  getContext().displayTopBottomResult(result);

}

/**
 * Displays a description of the results of the query.
 *
 * @param singleSelections The SingleSelectionsTemplate used by the query.
 *
 * @param topNBottom The TopBottomTemplate used by the query.
 *
```

```
   * @param resultDescr A String that contains a description of the query.
   */
  private void displayResultDescr(SingleSelectionTemplate singleSelections,
                                  TopBottomTemplate topNBottom,
                                  String resultDescr)
  {
    DataProvider dp = getContext().getDataProvider();

    // Get the short descriptions of the dimension members of the
    // SingleSelectionTemplate.
    StringBuffer shortDescrsForMemberVals =
                                  singleSelections.getMemberShortDescrs(dp);

    // Display the number of dimension members selected, the result description,
    // and the short descriptions of the single selection dimension members.
    println("\nThe " + Math.round(topNBottom.getN()) + resultDescr
            + shortDescrsForMemberVals +" are:\n");
  }

  /**
   * Runs the TopBottomTest application.
   *
   * @param args An array of String objects that provides the arguments
   *             required to connect to an Oracle Database instance, as
   *             specified in the Context11g class.
   */
  public static void main(String[] args)
  {
    new TopBottomTest().execute(args);
  }
}
```

The `TopBottomTest` program produces the following output.

```
The 10 products with the most units sold
for Asia Pacific, Direct Sales, 2001 are:

 1. Mouse Pad
 2. Unix/Windows 1-user pack
 3. Deluxe Mouse
 4. Laptop carrying case
 5. 56Kbps V.90 Type II Modem
 6. 56Kbps V.92 Type II Fax/Modem
 7. Keyboard Wrist Rest
 8. Internal - DVD-RW - 6X
 9. O/S Documentation Set - English
10. External - DVD-RW - 8X

The 5 products with the fewest units sold
for Asia Pacific, Direct Sales, 2000 are:

 1. Envoy External Keyboard
 2. O/S Documentation Set - Italian
 3. External 48X CD-ROM
 4. O/S Documentation Set - Spanish
 5. Internal 48X CD-ROM USB
```

```
The 7 products with the highest sales amounts
for Asia Pacific, Direct Sales, 2000 are:

 1. Sentinel Financial
 2. Sentinel Standard
 3. Envoy Executive
 4. Sentinel Multimedia
 5. Envoy Standard
 6. Envoy Ambassador
 7. 56Kbps V.90 Type II Modem

The 7 products with the lowest sales amounts
for Asia Pacific, Direct Sales, 2000 are:

 1. Envoy External Keyboard
 2. Keyboard Wrist Rest
 3. Mouse Pad
 4. O/S Documentation Set - Italian
 5. O/S Documentation Set - Spanish
 6. Standard Mouse
 7. O/S Documentation Set - French
```

A

Setting Up the Development Environment

This appendix describes the development environment for creating applications that use the OLAP Java API.

This appendix includes the following topics:

Overview

The OLAP Java API client software is a set of Java packages containing classes that implement the programming interface to Oracle OLAP. An Oracle Database with the OLAP option provides the OLAP Java API and other required class libraries as Java archive (JAR) files. As an application developer, you must copy the required JAR files to the computer on which you develop your Java application, or otherwise make them accessible to your development environment.

When a Java application calls methods of OLAP Java API objects, it uses the OLAP Java API client software to communicate with Oracle OLAP, which resides within an Oracle Database instance. The communication between the OLAP Java API client software and Oracle OLAP is provided through the Java Database Connectivity (JDBC) API, which is a standard Java interface for connecting to relational databases. Another required JAR file provides support for importing and exporting OLAP Java API metadata objects XML.

To use the OLAP Java API classes as you develop your application, import them into your Java code. When you deliver your application to users, include the OLAP Java API classes with the application. You must also ensure that users can access JDBC.

To develop an OLAP Java API application, you must have the Java Development Kit (JDK), such as one in Oracle JDeveloper or one from Sun Microsystems. Users must have a Java Runtime Environment (JRE) whose version number is compatible with the JDK that you used for development.

Required Class Libraries

Your application development environment must have the following files:

- The `olap_api.jar` file, which contains the OLAP Java API class libraries.
- The `ojdbc5.jar` file, which is an Oracle JDBC (Java Database Connectivity) library that contains classes required to connect to an Oracle Database instance.

The Oracle installation includes the JDBC file. You must use that JDBC file and not one from another Oracle product or from a product from another vender.

- The `xmlparserv2.jar` file, which contains classes that provide XML parsing support.
- The Java Development Kit (JDK) version 1.5. The Oracle Database installation does not provide the JDK. If you are using Oracle JDeveloper as your development environment, then the JDK is already installed on your computer. However, ensure that you are using the correct version of the JDK in JDeveloper. For information about obtaining and using some other JDK, see the Sun Microsystems Java Web site at

  `http://java.sun.com`

Obtaining the Class Libraries

Table A–1 lists the OLAP Java API and other JAR files that you must include in your application development environment. The table includes the locations of the files under the directory identified by the `ORACLE_HOME` environment variable on the system on which the Oracle Database is installed. You can copy these files to your application development computer, or otherwise include them in your development environment.

Table A–1 Required Class Libraries and Their Locations in the Oracle Installation

Class Library jar File	Location under ORACLE_HOME
`olap_api.jar`	`/olap/api/lib`
`ojdbc5jar`	`/jdbc/lib`
`xmlparserv2.jar`	`/lib`

B

SingleSelectionTemplate Class

This appendix contains the code for the `SingleSelectionTemplate` class. This class is used by the examples in Chapter 7, "Using a TransactionProvider", and Chapter 10, "Creating Dynamic Queries".

Code for the SingleSelectionTemplate Class

The following is the `SingleSelectionTemplate.java` class.

```
import oracle.olapi.data.cursor.CursorManager;
import oracle.olapi.data.cursor.ValueCursor;
import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.DynamicDefinition;
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.StringSource;
import oracle.olapi.data.source.SourceGenerator;
import oracle.olapi.data.source.Template;
import oracle.olapi.metadata.mdm.MdmAttribute;
import oracle.olapi.metadata.mdm.MdmDimensionMemberInfo;
import oracle.olapi.metadata.mdm.MdmHierarchy;
import oracle.olapi.metadata.mdm.MdmPrimaryDimension;
import oracle.olapi.transaction.TransactionProvider;
import oracle.olapi.transaction.NotCommittableException;
import oracle.olapi.transaction.metadataStateManager.MetadataState;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

/**
 * A Template that joins Source objects for selected members of
 * dimension hierarchies to a Source for a measure.
 */
public class SingleSelectionTemplate extends Template
{
  // Variable to store the DynamicDefinition.
  private DynamicDefinition dynamicDef;

  /**
   * Creates a SingleSelectionTemplate.
   */
  public SingleSelectionTemplate(Source measure, DataProvider dataProvider)
  {
    super(new SingleSelectionTemplateState(measure), dataProvider);
    dynamicDef = createDynamicDefinition(
```

```
                        new SingleSelectionTemplateGenerator(dataProvider));
}

/**
 * Gets the Source produced by the SingleSelectionTemplateGenerator
 * from the DynamicDefinition.
 */
public final Source getSource()
{
  return dynamicDef.getSource();
}

/**
 * Gets the Source for the measure stored by the SingleSelectionTemplateState.
 */
public Source getMeasure()
{
  SingleSelectionTemplateState state =
                             (SingleSelectionTemplateState) getCurrentState();
  return state.measure;
}

/**
 * Specifies the Source for the measure stored by the
 * SingleSelectionTemplateState.
 */
public void setMeasure(Source measure)
{
  SingleSelectionTemplateState state =
              (SingleSelectionTemplateState) getCurrentState();
  state.measure = measure;
  setCurrentState(state);
}

/**
 * Gets the List of MdmDimensionMemberInfo objects for the selected members
 * of the dimensions.
 */
public List getDimMemberInfos()
{
  SingleSelectionTemplateState state =
                             (SingleSelectionTemplateState) getCurrentState();
  return Collections.unmodifiableList(state.dimMemberInfos);
}

/**
 * Adds an MdmDimensionMemberInfo to the List of
 * MdmDimensionMemberInfo objects.
 */
public void addDimMemberInfo(MdmDimensionMemberInfo mdmDimMemberInfo)
{
  SingleSelectionTemplateState state =
                             (SingleSelectionTemplateState) getCurrentState();
  state.dimMemberInfos.add(mdmDimMemberInfo);
  setCurrentState(state);
}

/**
 * Changes the member specified for a dimension.
 */
```

```
public void changeSelection(MdmDimensionMemberInfo mdmDimMemberInfo)
{
  SingleSelectionTemplateState state =
                          (SingleSelectionTemplateState) getCurrentState();
  int i = 0;

  Iterator dimMemberInfosItr = state.dimMemberInfos.iterator();
  while (dimMemberInfosItr.hasNext())
  {
    MdmDimensionMemberInfo mdmDimMemberInfoInList =
                           (MdmDimensionMemberInfo) dimMemberInfosItr.next();
    MdmPrimaryDimension mdmPrimDim1 = mdmDimMemberInfo.getPrimaryDimension();
    MdmPrimaryDimension mdmPrimDim2 =
                               mdmDimMemberInfoInList.getPrimaryDimension();
    //String value = (String) valuesItr.next();
    if (mdmPrimDim1.getName().equals(mdmPrimDim2.getName()))
    {
      state.dimMemberInfos.remove(i);
      state.dimMemberInfos.add(i, mdmDimMemberInfo);
      break;
    }
    i++;
  }

  setCurrentState(state);
}

/**
 * Gets the short value description of the each of the dimension members
 * specified by the list of MdmDimensionMemberInfo objects and returns
 * the descriptions in a StringBuffer.
 */
public StringBuffer getMemberShortDescrs(DataProvider dp)
{
  boolean firsttime = true;

  List mdmDimMemInfoList = getDimMemberInfos();

  StringBuffer shortDescrForMemberVals = new StringBuffer(" ");
  Iterator mdmDimMemInfoListItr = mdmDimMemInfoList.iterator();

  while(mdmDimMemInfoListItr.hasNext())
  {
    MdmDimensionMemberInfo mdmDimMemInfo = (MdmDimensionMemberInfo)
                                         mdmDimMemInfoListItr.next();
    MdmPrimaryDimension mdmPrimDim = mdmDimMemInfo.getPrimaryDimension();
    MdmAttribute mdmShortDescrAttr =
                              mdmPrimDim.getShortValueDescriptionAttribute();
    Source shortDescrAttr = mdmShortDescrAttr.getSource();
    MdmHierarchy mdmHier = mdmDimMemInfo.getHierarchy();
    StringSource hierSrc = (StringSource) mdmHier.getSource();
    Source memberSel = hierSrc.selectValue(mdmDimMemInfo.getUniqueValue());
    Source shortDescrForMember = shortDescrAttr.joinHidden(memberSel);

    // Commit the current transaction.
    try
    {
      (dp.getTransactionProvider()).commitCurrentTransaction();
    }
```

```
      catch (Exception ex)
      {
        println("Could not commit the Transaction. " + ex);
      }
    }

      CursorManager cmngr = dp.createCursorManager(shortDescrForMember);
      ValueCursor valCursor = (ValueCursor) cmngr.createCursor();

      String shortDescrForMemberVal = valCursor.getCurrentString();

      if(firsttime)
      {
        shortDescrForMemberVals.append(shortDescrForMemberVal);
        firsttime = false;
      }
      else
      {
        shortDescrForMemberVals.append(", " + shortDescrForMemberVal);
      }
    }

    return shortDescrForMemberVals;
  }

  /**
   * Inner class that implements the MetadataState object for this Template.
   * Stores data that can be changed by its SingleSelectionTemplate.
   * The data is used by a SingleSelectionTemplateGenerator in producing
   * a Source for the SingleSelectionTemplate.
   */
  private static class SingleSelectionTemplateState
     implements MetadataState
  {
    public Source measure;
    public ArrayList dimMemberInfos;

    /**
     * Creates a SingleSelectionTemplateState.
     */
    public SingleSelectionTemplateState(Source measure)
    {
      this(measure, new ArrayList());
    }

    private SingleSelectionTemplateState(Source measure,
                                         ArrayList dimMemberInfos)
    {
      this.measure = measure;
      this.dimMemberInfos = dimMemberInfos;
    }

    public Object clone()
    {
      return new SingleSelectionTemplateState(measure,
                                              (ArrayList)
                                               dimMemberInfos.clone());
    }
  }
```

```
  /**
   * Inner class that implements the SourceGenerator object for this Template.
   * Produces a Source based on the data values of a SingleSelectionTemplate.
   */
  private static final class SingleSelectionTemplateGenerator
      implements SourceGenerator
  {
    DataProvider dp = null;

    /**
     * Creates a SingleSelectionTemplateGenerator.
     */
    public SingleSelectionTemplateGenerator(DataProvider dataProvider)
    {
      dp = dataProvider;
    }

    /**
     * Generates a Source for the SingleSelectionTemplate.
     */
    public Source generateSource(MetadataState state)
    {
      SingleSelectionTemplateState castState =
                                 (SingleSelectionTemplateState) state;
      Source result = castState.measure;

      Iterator dimMemberInfosItr = castState.dimMemberInfos.iterator();
      while (dimMemberInfosItr.hasNext())
      {
        MdmDimensionMemberInfo mdmDimMemInfo = (MdmDimensionMemberInfo)
                                        dimMemberInfosItr.next();
        MdmHierarchy mdmHier = mdmDimMemInfo.getHierarchy();
        StringSource hierSrc = (StringSource) mdmHier.getSource();
        Source memberSel = hierSrc.selectValue(mdmDimMemInfo.getUniqueValue());
        // Join the Source objects for the selected dimension members
        // to the measure.
        result = result.joinHidden(memberSel);
      }
      return result;
    }
  }
}
```

Index

D

E

F

N

O

P

Q

R

S

U

V

W

X