# Oracle® OLAP

Java API Developer's Guide

11*g* Release 1 (11.1)

**B28127-01**

July 2007

ORACLE®

Oracle OLAP Java API Developer's Guide, 11*g* Release 1 (11.1)

B28127-01



Primary Author: David McDermid

# Contents

## 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 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

*Oracle OLAP Java API Developer's Guide* is intended for Java programmers who are responsible for creating applications 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, with good usability, to the disabled community. 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.

### TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

## Related Documents

For more information, see these Oracle resources:

- *Oracle OLAP Java API Reference*
- *Oracle OLAP User's Guide*
- *Oracle OLAP DML Reference*
- *Oracle Database Data Warehousing Guide*

## 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 Oracle OLAP 11*g* Release 1 (11.1).

## 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 Expression Syntax
- Share a Connection Between Multiple Sessions
- Use Cost-Based Aggregation
- 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.

An application creates most metadata objects with findOrCreate methods of the owning object. For example, the following code finds the `MdmStandardDimension` named `PRODUCTS_AWJ` or creates a standard dimension with that name if it does not already exist. The `mdmDBSchema` object is the `MdmDatabaseSchema` that owns the dimension.

```
MdmStandardDimension prodDim =
            mdmDBSchema.findOrCreateStandardDimension("PRODUCTS_AWJ");
```

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 are instances of classes like `AW`, `MdmCube`, and `MdmPrimaryDimension`, those classes 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. An client application can set such restrictions by using the JDBC API to send SQL commands through the JDBC connection for the user session. See *Oracle OLAP User's Guide* for more information on object security.

### 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.

### 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.

### Use SQL Expression Syntax

With the classes in the `oracle.olap.syntax package`, an application can create Java objects that are based on SQL expressions, functions, operators, and conditions. The `SyntaxObject` class has static `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 exp = (StringExpression)
                    SyntaxObject.fromSyntax("'Hello world from syntax.'", mp);
StringExpression strExp = 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.

### Use Cost-Based Aggregation

Cost-based aggregation optimizes the aggregation and storage of summary data for an OLAP cube. The optimization can produce faster refresh and query operation performance.

Cost-based aggregation automatically determines the aggregate data within a cube to precompute and to store. It calculates the cost of computing the aggregate data in the cube and determines if it is better to precalculate the data and store it or to allow Oracle OLAP to calculate the data dynamically (on the fly).

An application can specify a percentage of the measure values that Oracle OLAP precomputes for a compressed cube. For a compressed and partitioned cube, an application can also specify the percentage of values to precompute for the top level of the partitioned dimension. When Oracle OLAP aggregates the measure values for the cube, it optimizes the aggregation based on the percentages specified. Methods of the `AWCubeOrganization` class get and set the percentage for precomputing.

### 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. The `DataProvider` class has a metadata reader mode. By default, the metadata reader recognizes Oracle OLAP 11*g* metadata. If an application that was created using a previous version of the OLAP Java API connects to an instance of Oracle Database 11*g* Release 1 (11.1) and creates a `DataProvider`, then Oracle OLAP automatically uses the appropriate metadata reader mode.

An application can specify a metadata reader mode with a property of a `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.

An application cannot mix metadata reader modes. The 11*g* metadata reader does not recognize earlier forms of metadata.

# 1

# Introduction to the OLAP Java API

This chapter introduces the Oracle OLAP Java API to application developers who plan to use it in their Java applications.

This chapter includes the following topics:



## OLAP Java API Overview

The OLAP Java API is a Java application programming interface (API) through which an application can implement a metadata model, define and build analytic workspaces, and access data for online analytical processing (OLAP). That data can be in relational database structures or it can be in analytic workspaces. The Java classes that implement the API are part of the Oracle OLAP component of Oracle Database.

The purpose of the OLAP Java API is to facilitate the development of OLAP applications, which allow users to create analytic workspaces and to dynamically select, aggregate, calculate, and perform other analytical tasks on data through a graphical user interface. Typically, the user interface of an OLAP application displays data in multidimensional formats, such as graphs and crosstabs.

In general, OLAP applications are developed within the context of business intelligence and data warehousing systems, and the features of the OLAP Java API are optimized for this type of application. With the OLAP Java API, a Java application can create and maintain analytic workspaces, and access, manipulate, and display relational or analytic workspace data in multidimensional terms. The OLAP Java API also makes it possible to define a query in a step-by-step process that allows for undoing individual query steps without reproducing the entire query. Such multistep queries are easy to modify and refine dynamically.

### Multidimensional Concepts and the OLAP Java API

Data warehousing and OLAP applications are based on a multidimensional view of data, and they work with queries that represent selections of data. The following definitions introduce concepts that reflect the multidimensional view and are basic to data warehousing, OLAP, and the OLAP Java API:

- Cube. A logical organization of multidimensional data that associates one or more measures with a set of dimensions. All of the measures are dimensioned by the same set of dimensions. An OLAP cube has edges and a body. Typically, the edges of a cube contain dimension member values, and the body of a cube contains measure values. For example, data on the quantity of product units sold can be organized into a cube whose edges contain values for members from the time, product, customer, and channel dimensions and whose body contains values from the measure of units sold.
- Measure. Data, usually numeric and additive, that can be examined and analyzed. Typically, a measure is categorized by one or more dimensions, and it is described as "dimensioned by" them.
- Dimension. A structure that categorizes data. Commonly-used dimensions are customers, products, and times. Typically, the members of a dimension are organized one or more hierarchies that have one or more levels. Sets of members of different dimensions identify measure values. 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, an application can formulate the query, "Did we sell more widgets in January or June?"
- Hierarchy. A logical structure that uses ordered levels or values as a means of organizing dimension members in parent-child relationships. Typically, in the user interface end users can expand or collapse the hierarchy by drilling down or up on the levels.
- Level. A component of a level-based hierarchy. For example, a time dimension might have a hierarchy that has members that represents data at the day, month, quarter, and year levels.
- Attribute. A descriptive characteristic of 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 a color attribute.
- Query. 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. The 11*g* release introduces a `Query` class in the `oracle.olapi.syntax` package. A `Query` represents a multi-row, multi-column result set that is similar to a relational table, a SQL `SELECT` statement, or an OLAP function. In this documentation, the general term *query* continues to refer to a `Source` object.

An edge is one side of a cube. The OLAP concept of an edge is not represented by a metadata object in the OLAP Java API, but is 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.

For more information about all of these concepts, see *Oracle OLAP User's Guide* and *Oracle Database Data Warehousing Guide*.

## What Type of Data Can an Application Access Through the OLAP Java API?

The OLAP Java API, as part of Oracle OLAP, 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 transaction processing. Warehouse data often conforms to a star schema, which represents a multidimensional data model. The star schema consists of one or more fact tables and one or more dimension tables that are related through foreign keys. 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. Once the data is organized in the warehouse, the database administrator can design an OLAP metadata model, map the logical metadata objects to data in the warehouse, and build an analytic workspace using Analytic Workspace Manager (AWM). Building the analytic workspace populates the storage structures with the data that the OLAP metadata objects represent. See *Oracle OLAP User's Guide* for information about supported data warehouse configurations and about creating an analytic workspace with AWM.

With the OLAP Java API, an application can also design the metadata model, map the logical objects to data in the warehouse, and build an analytic workspace. An application can then get the OLAP metadata objects created either by AWM 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 for which a database administrator has created an analytic workspace is the data store to which the OLAP Java API gives access. Of course, each user who accesses data through the OLAP Java API might have security restrictions that limit the scope of the data that he or she can access within the data store.

## What Can an Application Do with the OLAP Java API?

Through the OLAP Java API, an application can do the following:

- Establish a connection to an Oracle Database instance.
- Provide for multiple user sessions that share the same connection and the same cache of 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 what data is available for viewing or analysis.
- Create queries that specify and manipulate the data according to the needs of application users (for example, selecting, aggregating, and calculating data).
- Retrieve query results that are structured for display in a multidimensional format.
- Modify existing queries, rather than totally redefine them, as application users refine their analyses.

## Context for OLAP Java API Development

The OLAP Java API has all of the advantages of the Java environment. It is platform independent and it provides the benefits of an object-oriented API, such as abstraction, encapsulation, polymorphism, and inheritance. These strengths are built into the OLAP Java API and because the client application is written in Java, it can take advantage of them.

To work with the OLAP Java API, application developers should have familiarity with Java, object-oriented programming, relational databases, data warehousing, and multidimensional OLAP concepts.

# Sample Schema for OLAP Java API Examples

The examples of OLAP Java API code in this documentation are excerpts from example programs that query an analytic workspace named `GLOBAL_AWJ`. That analytic workspace is built from relational tables by the `BuildAW11g.java` example program. The relational tables are in the Global schema.

From the Oracle Technology Network (OTN) Web site, you can download a zip file that contains the SQL scripts that create the Global schema. To get that file, go to Sample Schemas for Documentation in the Documentation section. The Oracle Technology Network (OTN) Web site is at

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

On the OTN Web site, the complete code for the examples is available in the `examples.zip` file. That file is also in an Oracle Database installation. It is located in the same directory as the OLAP Java API class libraries. See Appendix A, "Setting Up the Development Environment" for that location.

The example programs are in a package structure that you can easily add to your development environment. At the top level of the package hierarchy is 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 persistent OLAP metadata objects that the `BuildAW11g.java` program creates 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 `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 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, which has the dollar amounts for the sales of product units. The cube is dimensioned by all four dimensions.
- PRICE_CUBE_AWJ, which is a cube that contains the measures UNIT_COST and UNIT_PRICE. UNIT_COST has the costs of a unit. UNIT_PRICE has the prices of a unit. The cube is dimensioned by the PRODUCT_AWJ and TIME_AWJ dimensions.

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

# Access to Data and Metadata Through the OLAP Java API

Oracle OLAP metadata objects describe the data that is available to the OLAP Java API through a connection to the database. The metadata objects record three things:

- The existence of sets of data. For example, a measure of unit price figures, dimensions of product and time member values, and attributes that contain information about the members of the dimensions all exist as named entities in the data store.
- The structure of the sets of data. For example, the Unit Price measure is dimensioned by products and times, an attribute is dimensioned by the dimension for which it records information, and the members of the dimensions are organized into hierarchical levels.
- The characteristics of the data. For example, the Unit Price measure contains numeric values that are specified by the dimension member values, the dimension members are unique String values, and the dimensions have attributes that provide additional information, such as a descriptive name for each dimension member.

In contrast, the fact that the price of a specific product in a specific month was 2,426.07 dollars is data, not metadata.

These examples distinguish between the metadata and the data for the measure of unit prices. The OLAP Java API makes a similar distinction between the metadata and the data for dimensions. For example, the fact that a product dimension exists and that the dimension members have text values is metadata. In contrast, the fact that the unique value of one of the dimension members is PRODUCT_PRIMARY::ITEM::ENV STD is data.

## MDM Model in the OLAP Java API

The OLAP Java API multidimensional metadata (MDM) model describes data in multidimensional terms, which are familiar to OLAP and data warehousing audiences. For example, it includes objects for cubes, measures, dimensions, hierarchies, and attributes.

The following are some of the Java classes that are supplied by the OLAP Java API in the implementation of the MDM model:

- MdmMetadataProvider
- MdmRootSchema

- MdmDatabaseSchema
- MdmCube
- MdmMeasure
- MdmDimension
- MdmDimensionLevel
- MdmHierarchy
- MdmHierarchyLevel
- MdmAttribute

An MdmMetadataProvider gives an application access to the MDM metadata objects that represent the OLAP metadata objects. To obtain the MDM metadata objects, an application uses the getRootSchema method of an MdmMetadataProvider. This method returns the MdmRootSchema, which is a container for the other accessible OLAP metadata objects. From the MdmRootSchema an application can get the MdmDatabaseSchema objects that are available. An MdmDatabaseSchema represents a schema in the relational database. As each database user owns a single relational schema, each user has a single MdmDatabaseSchema.

An MdmDatabaseSchema contains the other accessible metadata objects, such as cubes, measures, and dimensions. An MdmDatabaseSchema can have one or more subschemas. These subschemas are instances of the MdmOrganizationalSchema class, which corresponds to a measure folder in Analytic Workspace Manager.

From the MdmRootSchema, an application can also get all of the MdmCube, MdmMeasure, and MdmDimension objects that are available. From an MdmDatabaseSchema, an application can get the MdmCube, MdmMeasure, and MdmDimension objects that are owned by that schema.

The MdmDimension objects and MdmMeasure objects might be organized in a hierarchical tree, with MdmOrganizationalSchema subschemas nested under an MdmDatabaseSchema. An application can navigate through the objects owned by an MdmDatabaseSchema to discover the metadata objects that are available.

Chapter 2, "Understanding OLAP Java API Metadata", provides detailed information about the OLAP Java API metadata.

## Access to Data Through the OLAP Java API

An MdmMeasure or MdmDimension represents data in the data store. For example, an MdmMeasure object named mdmSales might represent a set of elements whose numeric values are dollar amounts for units sold, and an MdmDimension called mdmProdDim might represent a set of members whose text values are product identifiers. However, an application cannot create a query on the data using an MdmMeasure or MdmDimension. As metadata, MdmMeasure and MdmDimension objects provide descriptive information about data, but they do not provide the ability to construct a query that specifies the data. To select, calculate, and otherwise manipulate data for analysis, an application must create a query.

To create a query on the data for an MdmMeasure or MdmDimension, an application must first get the Source object for the MdmMeasure or MdmDimension by calling the getSource method of the metadata object. This method returns a Source object that the application can use to specify a query. The query defines a result set, and, in this case, the result set is the data for the MdmMeasure or MdmDimension.

In addition to representing the data for metadata objects, `Source` objects can represent the data for any query that an application creates. For example, a `Source` might specify a query for a selection of `MdmDimension` values (such as January, February, and March of the year 2002) or a calculation of the values of one `MdmMeasure` minus those of another (such as `unitPrice` minus `unitCost`). An application can use the powerful methods of the `Source` class and its subclasses to combine data in any way that the user requires.

One of the useful characteristic of `Source` objects is that they make no distinction between attributes, dimensions, and measures. The `Source` objects for all of them behave in the same way.

To retrieve the data specified by a `Source`, an application creates a `Cursor` for that `Source`. The application then uses this `Cursor` to request and retrieve the data from the data store. When an application makes a request for data, it can specify the typical amount of data that it requires at a given time (for example, enough to fill a 40-cell table on the screen). Oracle OLAP then handles the issues related to efficient retrieval. The application does not need to manage the timing, sizing, and caching of the data blocks that it retrieves through the OLAP Java API.

## Unique and Local Dimension Values

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 OLAP Java API uses a three-part format to specify the hierarchy, the level, and the value of a dimension member, and thus identify a unique value in the hierarchy. 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 in the `YEAR` level of the `CALENDAR_YEAR` hierarchy of the `TIME_AWJ` dimension:

```
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.

The OLAP Java API has classes and methods that you can use to get the local values of dimension members. The `MdmPrimaryDimension` class has a method for getting an `MdmAttribute` that records the local values for the members of the hierarchies that are components of the `MdmPrimaryDimension`, and the `MdmDimensionMemberInfo` class has methods for getting the local or unique values for a member of a hierarchy or a level.

## User Connection Requirements

In addition to ensuring that data and metadata have been prepared appropriately, an application developer must ensure that application users can make a connection to the data store through the OLAP Java API and that users have database privileges that give them access to the data. For information about setting up for such connections, see the *Oracle OLAP User's Guide*.

# OLAP Java API Client Software

The OLAP Java API client software is a set of Java packages containing classes that implement the programming interface to Oracle OLAP. An application creates objects of these classes and calls their methods to create or discover metadata, specify queries, and retrieve data.

When a Java application calls methods of objects of OLAP Java API classes, 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.

## Requirements for Using the OLAP Java API Client Software

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.

In order 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.

For information about Java version requirements and about setting up the OLAP Java API client software, see Appendix A, "Setting Up the Development Environment". For detailed information about the OLAP Java API classes and methods, see *Oracle OLAP Java API Reference* and the subsequent chapters of this guide.

# Tasks That an OLAP Java API Application Performs

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

1. Connects to the data store
2. Creates or discovers metadata objects
3. Defines and builds an analytic workspace, 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.

## Task 1: Connect to the Data Store

An application connects to the data store by identifying some information about the target Oracle Database instance and specifying this information in a JDBC connection method.

For more information about connecting, see Chapter 3, "Discovering Metadata".

## Task 2: Create or Discover Metadata Objects

Having established a connection, the application creates a `DataProvider` and uses it to get an `MdmMetadataProvider`. The `MdmMetadataProvider` gives access to all of the metadata objects in the data store.

To discover the available metadata, an application uses the `getRootSchema` method of the `MdmMetdataProvider` to obtain the `MdmRootSchema` object, which contains all of the metadata objects. The application then gets the `MdmDatabaseSchema` object or objects that the current user has permission to access.

From an `MdmDatabaseSchema`, the application can discover the existing metadata objects that are owned by schema or 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.

If an application creates a new metadata object that represents data, it must specify an `Expression` that maps the metadata object to a relational source table or that Oracle OLAP uses to generate the data. For information on creating metadata, see Chapter 4, "Creating Metadata and Analytic Workspaces".

From a top-level metadata objects, such as an analytic workspace, cube, or dimension, an application can get the objects that belong to it. For example, from an `MdmPrimaryDimension`, an application can get the hierarchies, levels, and attributes that are associated with it. Having determined the metadata objects that it has to work with, the application 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 an application can discover the available metadata, see Chapter 3, "Discovering Metadata".

## Task 3: Select and Calculate Data Through Queries

A typical OLAP application constructs queries against the data store. The 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 on 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 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 4: Retrieve Query Results

When users of an OLAP 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, the application creates a `Cursor` for the `Source` that specifies the query.

An application can also get the SQL that Oracle OLAP generates for a query. To do so, the application creates a `SQLCursorManager` for the `Source` instead of creating a `Cursor`. The `generateSQL` method of the `SQLCursorManager` returns the SQL specified by the `Source`. The application can then retrieve the data by methods 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`. An application can create a query by joining the `Source` objects for the dimensions to the `Source` for the measure. The query has the `Source` for the measure as the base and it has the `Source` objects for the dimensions as outputs.

A `Cursor` for the query `Source` has the same structure as the `Source`; that is, the `Cursor` has base values that are the measure data and the `Cursor` 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`, the application 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 metadata objects. This chapter includes the following topics:



## Overview of the OLAP Java API Metadata

Chapter 1 described the OLAP dimensional data model and briefly mentioned some of the classes in the OLAP Java API that represent OLAP objects. This chapter describes the OLAP Java API classes that represent metadata objects and discusses how an application uses them.

The OLAP Java API provides a metadata model called MDM (multidimensional metadata). The classes that implement the model are in the `oracle.olapi.metadata.mdm` package.

An application can use classes in that package to discover existing metadata objects or to create new ones. To have access to the metadata objects, the user specified in the connection to the Oracle Database instance must have the required system privileges. Before an application can create metadata objects, the data structures in the Oracle Database instance must conform to OLAP requirements. Those system privileges and requirements are described in *Oracle OLAP User's Guide*.

Some of the classes in the `oracle.olapi.metadata.mdm` package directly correspond to OLAP metadata objects. The following table presents some of these correspondences.

| Oracle OLAP Metadata Objects | MDM Metadata Objects |
|---|---|
| Cube | `MdmCube` |
| Measure | `MdmBaseMeasure` |
| Dimension | `MdmTimeDimension` or `MdmStandardDimension` |

| Oracle OLAP Metadata Objects | MDM Metadata Objects |
|---|---|
| Hierarchy | `MdmLevelHierarchy` or `MdmValueHierarchy` |
| Level | `MdmDimensionLevel` and `MdmHierarchyLevel` |
| Attribute | `MdmAttribute` |
| Measure folder | `MdmOrganizationalSchema` |

Most of the classes in the `oracle.olapi.metadata.mdm` package are subclasses of `MdmObject`. The remainder of this chapter describes some of those classes.

# MdmSchema Classes

`MdmSchema` objects are containers for `MdmCube`, `MdmMeasure`, `MdmDimension`, other `MdmSchema` objects. An `MdmSchema` has methods for getting all of the `MdmMeasure`, `MdmPrimaryDimension`, and `MdmSchema` objects that it contains. This section describes the subclasses of the `MdmSchema` class: `MdmRootSchema`, `MdmDatabaseSchema`, and `MdmOrganizationalSchema`.

## MdmRootSchema Class

Data that is accessible through the OLAP Java API is arranged under a top-level `MdmSchema`, which is an instance of `MdmRootSchema`. Under the `MdmRootSchema` are `MdmDatabaseSchema` objects.

From the `MdmRootSchema`, an application can get `List` objects that contain all of the `MdmCube`, `MdmDatabaseSchema`, `MdmDimension`, and `MdmMeasure` objects that are in the data store. The `MdmRootSchema` also has a method for getting the `MdmMeasureDimension`, whose members are all of the `MdmMeasure` objects in the data store.

## MdmDatabaseSchema Class

The `MdmRootSchema` has one `MdmDatabaseSchema` object for each Oracle Database user. The `MdmDatabaseSchema` corresponds to the relational schema that is owned by the user. The name of the `MdmDatabaseSchema` is the same as the name of the user.

The `MdmRootSchema` has a method for getting a list of all of the available `MdmDatabaseSchema` objects. It also has a method for getting an `MdmDatabaseSchema` by name.

An `MdmDatabaseSchema` owns the top-level OLAP metadata objects that are associated with a database user. Top-level objects include `MdmCube` and `MdmDimension` objects. An `MdmDatabaseSchema` can have one or more subschemas that group `MdmCube`, `MdmMeasure`, and `MdmDimension` objects. The subschemas are instances of `MdmOrganizationalSchema`.

An `MdmDatabaseSchema` has methods for finding top-level objects by name or creating the object if it does not already exist. Creating objects is described in Chapter 3.

### MdmOrganizationalSchema Class

An MdmOrganizationalSchema is equivalent to a folder or directory that contains associated items. It does not correspond to a relational schema in the Oracle database. Instead, it corresponds to an Oracle OLAP measure folder, which can include data from several relational schemas.

An MdmOrganizationalSchema can contain MdmCube, MdmMeasure, and MdmDimension objects. It can also have other MdmOrganizationalSchema objects as nested subschemas.

The MdmDatabaseSchema class has a method for getting a list of all of the MdmOrganizationalSchema objects that it owns. It also has methods for finding an MdmOrganizationalSchema by name or creating the object if it does not already exist.

## MdmSource Classes

MdmSource objects are the metadata objects that represent data that is available to an application. Subclasses of MdmSource include MdmDimension, MdmDimensionedObject, and MdmTable.

With the getSource method of an MdmSource, an application gets a Source object that it can use to create a query. The following line of code gets the Source for an MdmStandardDimension called mdmProductDim.

```
Source productDim = mdmProductDim.getSource();
```

A Source that is the result of the getSource method of an MdmSource is called a primary Source. An application derives new Source objects from this primary Source as it selects, calculates, and otherwise manipulates the data. When the application derives a Source that represents the query that it wants to make, it creates a Cursor for the Source. The Cursor retrieves 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".

## MdmDimension Classes

MdmDimension is an abstract subclass of MdmSource that 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.

An MdmDimension can have one or more MdmAttribute objects. An MdmAttribute maps the value of each member of the MdmDimension to a value representing some characteristic of the member value. To obtain the MdmAttribute objects for an MdmDimension, call the getAttributes method or the methods that return specific attributes, such as the getHierarchyAttribute or the getParentAttribute method.

## MdmPrimaryDimension Classes

MdmPrimaryDimension 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 has one or more component MdmDimensionLevel that organize the dimension members into levels. It also has one or more 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 the all of the MdmPrimaryDimension objects that belong to an MdmDatabaseSchema or an MdmOrganizationalSchema by calling the getDimensions method of the object. An MdmDatabaseSchema has methods for finding an MdmTimeDimension or and MdmStandardDimension by name or creating the object if it does not already exist.

## MdmSubDimension Classes

An MdmPrimaryDimension typically organizes the dimension members into hierarchical parent-child relationships. These hierarchical relationships are implemented by the subclasses of the abstract class MdmSubDimension. The subclasses of MdmSubDimension are MdmDimensionLevel, MdmHierarchy and MdmHierarchyLevel.

### MdmDimensionLevel Class

An MdmDimensionLevel represents a set of dimension members that are at the same hierarchical level. MdmHierarchy objects organize dimension levels into a hierarchy. An MdmDimensionLevel is associated with an MdmHierarchy by an MdmHierarchyLevel object.

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

### MdmHierarchy Class

MdmHierarchy is an abstract subclass of MdmSubDimension. An MdmHierarchy organizes the members of MdmDimensionLevel objects into a hierarchical structure. An MdmPrimaryDimension can have more than one hierarchy. For example, an MdmTimeDimension dimension might have two hierarchies, 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. The MdmHierarchyLevel for the lowest level of each of these hierarchies associates the same MdmDimensionLevel with each hierarchy.

The parent-child relationships of an MdmHierarchy are recorded in a parent MdmAttribute, which you can get by calling the getParentAttribute method of the MdmHierarchy. The ancestor-descendent relationships are specified 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 owns. It also has methods for finding an MdmLevelHierarchy or MdmValueHierarchy by name or creating the object if it does not already exist.

### MdmLevelHierarchy Class

MdmLevelHierarchy is a subclass of MdmHierarchy. An MdmLevelHierarchy has parent-child relationships that are defined between the values of the members at different levels. The different levels of an MdmLevelHierarchy are represented by MdmHierarchyLevel objects. An MdmLevelHierarchy can have up to 31 component MdmHierarchyLevel objects. An MdmLevelHierarchy has a tree-like structure. The members at the lowest level of the hierarchy are the leaves, and the members at higher levels are nodes. Nodes have children; leaves do not.

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. Each member, except those at the highest level, can have a parent, and each member, except those at the lowest level, can have one or more children. The parent and children of a member of an MdmHierarchyLevel are in other MdmHierarchyLevel objects. 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 owns. It also has a method for finding an MdmHierarchyLevel by name or creating the object if it does not already exist.

### MdmValueHierarchy Class

MdmValueHierarchy is the other subclass of MdmHierarchy. The members of an MdmValueHierarchy are not in an MdmDimensionLevel. The members typically come from a column in a relational embedded totals (ET) view. You can get the name of the view with the getETViewName method of the MdmValueHierarchy.

The MdmValueHierarchy defines the parent-child relationships of its members by the values of the members. 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.

### MdmHierarchyLevel Class

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

# MdmDimensionedObject Classes

MdmDimensionedObject is an abstract subclass of MdmSource that represents objects that have values that are specified by members of one or more dimensions. An MdmDimensionedObject has MdmDimensionality objects that associate the MdmPrimaryDimension objects with it. The subclasses of MdmDimensionedObject are MdmCube, MdmMeasure, and MdmAttribute.

## 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.

An MdmCube has an associated CubeMap and a CubeOrganization. The CubeMap has MeasureMap and CubeDimensionality objects that map the measures and dimensions of the cube to data sources. The CubeOrganization deploys the cube in an analytic workspace or as a relational database object. An MdmCube also has a ConsistentSolveSpecification object, which specifies how Oracle OLAP aggregates the measures of the cube.

## MdmMeasure Class

An MdmMeasure is an abstract class for an object that represents a set of data that is organized by one or more MdmDimension 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 MdmDimension 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 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 typically gets leaf-level data from a column in a fact table. The node-level data is calculated by Oracle OLAP. An MdmDerivedMeasure has values that result from mathematical calculations or a data transformations that Oracle OLAP performs on values from MdmBaseMeasure objects.

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

The MdmDimension 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 a node level of an MdmHierarchy.
- Values assigned by an MdmModel for a custom dimension member.

As an example, imagine an MdmMeasure 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 node members of the dimension it represents.

A unique combination of two members, one from mdmTimeDim and one from mdmProdDim, identifies each mdmUnitCost element, and every possible combination 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 node members (for example, a particular product family and a particular quarter). Still other mdmUnitCost elements are identified by a mixture of leaf and node 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 node member, Oracle OLAP provides the values. These higher-level values represent aggregated, or rolled-up, data.

Thus, the data represented by an MdmMeasure 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.

## MdmAttribute Class

MdmAttribute is an abstract subclass of MdmDimensionedObject. An MdmAttribute represents a particular characteristic of the members of an MdmDimension. An MdmAttribute relates a value to a member of the MdmDimension.

For example, mdmCustDim is the MdmPrimaryDimension for the Customer dimension. The MdmPrimaryDimension has a hierarchy that has levels that are based on shipment origination and destination values. The MdmAttribute returned by the getShortValueDescriptionAttribute method of mdmCustDim relates a short description to each the member of the dimension. The elements of the MdmAttribute have String values such as Europe, Italy, or Computer Services Athens.

The elements of an MdmAttribute might have String values (such as Italy), numeric values (such as 45), or objects (such as MdmHierarchyLevel objects).

Like an MdmMeasure, an MdmAttribute has elements that are organized by the MdmDimension associated with it. Sometimes an MdmAttribute does not have a value for every member of the MdmDimension. For example, an MdmAttribute that records the name of a contact person might have values only for the Ship To and Warehouse levels of the Shipments hierarchy of the mdmCustDim dimension, because

contact information does not apply to the higher Region and Total Customers levels. If an MdmAttribute does not apply to a member of an MdmDimension, then the MdmAttribute element value for that member is null.

An MdmAttribute object can provide a mapping that is one-to-many, rather than one-to-one. Therefore, a member in an MdmDimension might map to a whole set of MdmAttribute elements. For example, the MdmAttribute that serves as the ancestors attribute for an MdmHierarchy maps each MdmHierarchy member to the set of MdmHierarchy members that are the ancestors of it.

The following table lists the values of a Cursor for a Source object that represents the members of a hierarchy of an MdmPrimaryDimension of products. The table also lists the values of the Source objects for two MdmAttribute objects that are dimensioned by the MdmPrimaryDimension. One attribute is the short description attribute for the dimension. Each member of the dimension has a related short description. The other is an attribute that relates a package to the values of some of the members at the lowest level of the hierarchy. The values of the package MdmAttribute are null for the aggregate Total Product, Class, and Family levels and for unassigned Item level values. In the table, null values appear as NA. In the first column of the table, the value does not include the PRODUCT_PRIMARY hierarchy component of the unique dimension member value.

| Hierarchy 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 - DVD-RW - 8X | Laptop Value Pack |
| ITEM::INT CD USB | Internal 48X CD-ROM USB | NA |
| ITEM::INT RW DVD | Internal - DVD-RW - 6X | Multimedia |
| … | … | … |

# Data Type and Type of MDM Metadata Objects

All MdmSource objects have the following two basic characteristics:

- Data type
- Type

MdmDimensionCalculationModel objects also have a data type and a type. MdmDimensionedObjectModel objects have a type but not a data type.

## Data Type of MDM Metadata Objects

The concept of data type is a familiar one in computer languages and database technology. It is common to categorize data into types such as integer, Boolean, and String.

The OLAP Java API implements the concept of data type through the FundamentalMetadataObject and FundamentalMetadataProvider classes.

Every data type recognized by the OLAP Java API is represented by a `FundamentalMetadataObject`, and you obtain this object by calling a method of a `FundamentalMetadataProvider`.

The following table lists the most familiar OLAP Java API data types. For each data type, the table presents a description of the `FundamentalMetadataObject` that represents the data type and the name of the method of `FundamentalMetadataProvider` that returns the object. The OLAP Java API data types appear in regular typeface, instead of monospace typeface, to distinguish them from `java.lang` data type classes.

| OLAP Java API Data Type | Description of the `FundamentalMetadataObject` | Method of `FundamentalMetadataProvider` |
|---|---|---|
| Boolean | Represents the data type that corresponds to the Java `boolean` data type. | `getBooleanDataType` |
| Date | Represents the data type that corresponds to the Java `Date` class. | `getDateDataType` |
| Double | Represents the data type that corresponds to the Java `double` data type. | `getDoubleDataType` |
| Float | Represents the data type that corresponds to the Java `float` data type. | `getFloatDataType` |
| Integer | Represents the data type that corresponds to the Java `int` data type | `getIntegerDataType` |
| Short | Represents the data type that corresponds to the Java `short` data type. | `getShortDataType` |
| String | Represents the data type that corresponds to the Java `String` class. | `getStringDataType` |

In addition to these familiar data types, the OLAP Java API includes two generalized data types (which represent groups of the familiar data types) and two data types that represent the absence of values. The following table lists these additional data types.

| OLAP Java API Data Type | Description of the `FundamentalMetadataObject` | Method of `FundamentalmetadataProvider` |
|---|---|---|
| Number | Represents a general data type that includes any or all of the following OLAP Java API numeric data types: Double, Float, Integer, and Short | `getNumberDataType` |
| Value | Represents a general data type that includes any or all of the OLAP Java API data types. | `getValueDataType` |
| Empty | Represents no data, for example when an MdmSource has no elements at all defined for it. | `getEmptyDataType` |

| OLAP Java API Data Type | Description of the `FundamentalMetadataObject` | Method of `FundamentalmetadataProvider` |
|---|---|---|
| Void | Represents null data, for example when an MdmSource has a single element that has a null value. | `getVoidDataType` |

When an MDM metadata object, such as an `MdmMeasure`, has a given data type, this means that each of the elements of it conforms to that data type. If the data type is numeric, then the elements also conform to the generalized Number data type, as well as to the specific data type (Double, Float, Integer, or Short). The elements of any MDM metadata object conform to the Value data type, as well as to their more specific data type, such as Integer or String.

If the elements of an object represent a mixture of several numeric and non-numeric data types, then the data type is only Value. The object has no data type that is more specific than that.

The MDM metadata objects for which data type is relevant are `MdmDimensionCalculationModel` objects and `MdmSource` objects, such as `MdmMeasure`, `MdmLevelHierarchy`, and `MdmHierarchyLevel`. The typical data type of an `MdmMeasure` is one of the numeric data types; the data type of an `MdmLevelHierarchy` or `MdmHierarchyLevel` is always String.

An `MdmPrimaryDimension` has a set of `MdmDimensionCalculationModel` objects, each of which has a different data type. If an `MdmDimensionCalculationModel` has an `Assignment`, then Oracle OLAP assigns the specified value to measures that have the same data type as the `MdmDimensionCalculationModel`. For example, the data type of the `MdmDimensionCalculationModel` returned by the `getNumberCalcModel` method of an `MdmStandardDimension` is the `FundamentalMetadataObject` for the Number data type. An `Assignment` specified by that `MdmDimensionCalculationModel` applies only to a measure that has a Number data type and that is dimensioned by the `MdmStandardDimension`.

## Getting the Data Type of an MdmSource

To find the data type of an `MdmSource` or `MdmDimensionCalculationModel`, call the `getDataType` method of it. That method returns a `FundamentalMetadataObject`.

To find the OLAP Java API data type that is represented by the returned `FundamentalMetadataObject`, you could compare it to the `FundamentalMetadataObject` for each OLAP Java API data type. That is, you compare it to the return value of each of the data type methods in `FundamentalMetadataProvider`.

The following sample method returns a `String` that indicates the data type of an `MdmSource`. Note that this code gets the `FundamentalMetadataProvider` by calling a method of a `DataProvider`. Getting a `DataProvider` is described in Chapter 3, "Discovering Metadata".

***Example 2–1 Getting the Data Type of an MdmSource***

```
public String getDataType(DataProvider dp, MdmSource mdmSource)
{
   String theDataType = null;
```

```
FundamentalMetadataProvider fmp =
                            dp.getFundamentalMetadataProvider();

if (fmp.getBooleanDataType() == mdmSource.getDataType())
    theDataType = "Boolean";
 else if (fmp.getDateDataType() == mdmSource.getDataType())
    theDataType = "Date";
 else if (fmp.getDoubleDataType() == mdmSource.getDataType())
    theDataType = "Double";
 else if (fmp.getFloatDataType() == mdmSource.getDataType())
    theDataType = "Float";
 else if (fmp.getIntegerDataType() == mdmSource.getDataType())
    theDataType = "Integer";
 else if (fmp.getShortDataType() == mdmSource.getDataType())
    theDataType = "Short";
 else if (fmp.getStringDataType() == mdmSource.getDataType())
    theDataType = "String";
 else if (fmp.getNumberDataType() == mdmSource.getDataType())
    theDataType = "Number";
 else if (fmp.getValueDataType() == mdmSource.getDataType())
    theDataType = "Value";

 return theDataType;
 }
```

## Type of MDM Metadata Objects

An MDM metadata object, such as an `MdmSource`, is a collection of elements. The type of the object (as opposed to its data type) is another metadata object from which the metadata object draws elements. In other words, the elements of a metadata object correspond to a subset of the elements of the type object. There can be no element in the metadata object that does not match an element of the type.

Consider the following example of a `MdmPrimaryDimension` called `mdmCustDim`, which has the OLAP Java API data type of String. The `mdmCustDim` dimension has a hierarchy, which is an `MdmLevelHierarchy` object called `mdmShipments`, which in turn has levels, which are `MdmHierarchyLevel` objects. The `MdmLevelHierarchy` and the `MdmHierarchyLevel` objects represent subsets of the members of the `MdmPrimaryDimension`. In the following list, the hierarchy and the levels are indented under the `MdmPrimaryDimension` to which they belong.

```
mdmCustDim
    mdmShipments
       mdmTotalCust
       mdmRegion
       mdmWarehouse
       mdmShipTo
```

Because of the hierarchical structure, `mdmWarehouse` (for example) derives members from the members of `mdmShipments`. That is, the set of members for `mdmWarehouse` corresponds to a subset of members from `mdmShipments`, and `mdmShipments` is the type of `mdmWarehouse`.

Similarly, `mdmShipments` is a component hierarchy of `mdmCustDim`. Therefore, `mdmShipments` derives members from `mdmCustDim`, which is the type.

However, `mdmCustDim` is not a component of any other object. It represents the entire dimension. The pool of elements from which `mdmCustDim` derives members is the entire set of possible String values. Therefore, the type of `mdmCustDim` is the

FundamentalMetadataObject that represents the OLAP Java API String data type. In the case of mdmCustDim, the type and the data type are the same.

The following list presents the types that are typical for the most common MdmSource objects:

- The type of an MdmHierarchyLevel is the MdmLevelHierarchy to which it belongs.
- The type of a MdmHierarchy is the MdmPrimaryDimension to which it belongs.
- The type of an MdmPrimaryDimension is the FundamentalMetadataObject that represents the OLAP Java API data type of the MdmPrimaryDimension. Typically, this is the String data type.
- The type of an MdmMeasure is the FundamentalMetadataObject that represents the OLAP Java API data type of the MdmMeasure. Typically, this is one of the OLAP Java API numeric data types.

An MdmModel also has a type, which is the Source from which Oracle OLAP draws the values that the MdmModel assigns. For example, the type of the MdmDimensionedObjectModel for the MdmAttribute for the short value description attribute of the Product dimension is the Source for the FundamentalMetadataObject for the String data type because the values of that attribute are String objects.

## Getting the Type of an MdmSource

To find the type of an MdmSource, call the getType method of the MdmSource. That method returns the object that is the type of the MdmSource object.

Example 2–2 obtains the type of mdmWarehouse, which is an instance of an MdmHierarchyLevel. It also gets and displays the identifier of the object returned by the getType method, which is the hierarchy to which the level belongs.

***Example 2–2 Getting the Type of an MdmSource***

```
MetadataObject mdmWarehouseType = mdmWarehouse.getType();
println(mdmWarehouseType.getID());
```

The example displays the following:

```
GLOBAL.CUSTOMER_AWJ.SHIPMENTS
```

# Other MDM Metadata Objects

Other classes in the oracle.olapi.metadata.mdm package include MdmDescriptionType and MdmModel.

## MdmDescriptionType Class

An MdmDescriptionType represents a type of description for an MdmDescription. Static methods of the MdmDescriptionType class return standard types of descriptions such as name, plural name, description, long description, and others. An application can create other types of descriptions by constructing new MdmDescriptionType objects.

An application specifies an MdmDescriptionType when calling the findOrCreateDescription method of an MdmObject, which returns an MdmDescription.

## MdmModel Class

The MdmModel class implements the Model interface for MdmSource objects. Because a Model is closely is associated with a Source, the Model interface is in the oracle.olapi.data.source package. The Model interface is discussed in the topic "Model Objects and Source Objects" in Chapter 5, "Understanding Source Objects".

The MdmModel classes are an advanced feature of the OLAP Java API. When an application creates an MdmMember object, Oracle OLAP automatically creates an MdmModel for the MdmMember or adds information to an existing MdmModel object.

You can get an MdmModel for an MdmPrimaryDimension or an MdmDimensionedObject and use the MdmModel to specify the calculation of a value for a dimension member and the assignment of that value to the Source for a measure or attribute that is dimensioned by the dimension.

The subclasses of MdmModel are MdmDimensionCalculationModel and MdmDimensionedObjectModel. An MdmDimensionedObject object has an associated MdmDimensionedObjectModel that represents the assignment of zero or more values for the Source for the MdmDimensionedObject. You can get the MdmDimensionedObjectModel for an MdmDimensionedObject by calling the getModel method of the MdmDimensionedObject. The concrete subclasses of MdmDimensionedObjectModel are MdmAttributeModel and MdmMeasureModel.

An MdmDimensionCalculationModel assigns values for a measure of a particular data type. An MdmPrimaryDimension object has MdmDimensionCalculationModel objects for the OLAP Java API data types Boolean, Date, Number, and String. The MdmMeasureDimension subclass of MdmPrimaryDimension has a MdmDimensionCalculationModel for the Value data type, as well. You get an MdmDimensionCalculationModel for a specific data type by calling a method of an MdmPrimaryDimension, such as the getStringCalcModel method. Calling the getModel method of an MdmPrimaryDimension returns null.

The subclasses of MdmSubDimension, and the MdmStandardMember and MdmTimeMember classes, do not have associated MdmModel objects. Calling the getModel method of an MdmSubDimension, MdmStandardMember, or MdmTimeMember returns null.

# 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

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 make an OLAP Java API connection to an Oracle Database instance, 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 make a connection, perform the following steps:

1. Load the JDBC driver for the connection.
2. Get a JDBC `OracleConnection` from the `DriverManager`.

3. Create a `DataProvider` and a `UserSession`.

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

### Loading the JDBC Driver

The following code loads a JDBC driver and registers it with the JDBC `DriverManager`.

***Example 3–1 Loading the JDBC Driver for a Connection***

```
try
{
  Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException e)
{
  System.out.println("Could not load the JDBC driver. " + e);
}
```

After the driver is loaded, you can use the `DriverManager` object to make a connection. For more information about loading Oracle JDBC drivers, see *Oracle Database JDBC Developer's Guide and Reference*.

### Getting a Connection from the DriverManager

The following code gets a JDBC `OracleConnection` object from the `DriverManager`.

***Example 3–2 Getting a JDBC OracleConnection***

```
String url = "jdbc:oracle:thin:@myhost:1521:orcl";
String user = "global";
String password = "global";
oracle.jdbc.OracleConnection conn = null;
try
{
  conn = (oracle.jdbc.OracleConnection)
         java.sql.DriverManager.getConnection(url, user, password);
}
catch(SQLException e)
{
  System.out.println("Connection attempt failed. " + e);
}
```

This example connects the user `global`, who has the password `global`, to a database with the SID (system identifier) `orcl`. The connection is made through TCP/IP listener port `1521` of host `myhost`. The connection uses the Oracle JDBC thin driver.

There are many ways to specify your connection characteristics using the `getConnection` method. See *Oracle Database JDBC Developer's Guide and Reference* for details.

After you have the `OracleConnection` object, you can create the required OLAP Java API `DataProvider` and from it get the `TransactionProvider`.

### Creating a DataProvider and a UserSession

The following code creates a `DataProvider` and a `UserSession`. The `conn` object is the JDBC connection from Step 2.

***Example 3–3 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". 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 3–4 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, as described in Chapter 4, "Creating Metadata and Analytic Workspaces". 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 3–5 creates an `MdmMetadataProvider`. In the example, `dp` is the `DataProvider`.

***Example 3–5 Creating an MdmMetadataProvider***

```
MdmMetadataProvider mp = null;
try
{
  mp = (MdmMetadataProvider) dp.getDefaultMetadataProvider();
}
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 3–6 demonstrates getting the MdmRootSchema, the MdmDatabaseSchema objects under it, and any MdmOrganizationalSchema objects under them.

***Example 3–6 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 3–7 demonstrates getting the MdmDatabaseSchema for the GLOBAL user.

***Example 3–7 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 3–8 gets the dimensions and measures from the MdmDatabaseSchema from Example 3–7. It displays the name of each dimension and measure.

***Example 3–8 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 3–8 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 then gets the dimensions and measures of the cube. Example 3–9 gets a MdmCube from the MdmDatabaseSchema of Example 3–7 and displays the names of the dimensions and measures associated with it using the getNames method of Example 3–8.

***Example 3–9 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 3–9 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 Owned 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 by the `MdmHierarchyLevel` objects by calling the `getHierarchyLevels` method of the `MdmLevelHierarchy`.

Example 3–10 gets an `MdmPrimaryDimension` from the `MdmDatabaseSchema` of Example 3–7 and displays the names of the hierarchies and the levels associated with them.

***Example 3–10 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 3–10 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 3–11 demonstrates getting the `MdmAttribute` objects for an `MdmPrimaryDimension`. It also gets the parent attribute. The example displays the names of the `MdmAttribute` objects.

***Example 3–11 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 3–11 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 3–12 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 the "Creating Attributes" topic.

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 4–8, respectively.

***Example 4–2 Creating and Deploying an MdmStandardDimension***

```
MdmStandardDimension mdmChanDim =
                     mdmDBSchema.findOrCreateStandardDimension("CHANNEL_AWJ");
AWPrimaryDimensionOrganization awChanDimOrg  =
                                    mdmChanDim.createAWOrganization(aw, true);

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 4–5.

***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.

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 hierLevelName : dimLevelNames)
  {
    MdmHierarchyLevel mdmHierLevel =
     mdmLevelHier.findOrCreateHierarchyLevel(mdmLevelHier.getPrimaryDimension()
                 .findOrCreateDimensionLevel(dimLevelName));
    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 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 4–5 creates a long description attribute for the `CHANNEL_AWJ` dimension and makes it visible on the dimension.

***Example 4–5 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 4–6 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 4–7 and Example 4–8, respectively.

***Example 4–6 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.createAWOrganization(aw, true);
  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 4–7 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.createCubeMap();
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 4–8 gets the `TransactionProvider` from the `DataProvider` for the session and commits the current `Transaction`.

***Example 4–8 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 4–9 exports metadata objects to an XML template and saves it in a file. The following code calls the `exportToXML` method. The `aw` object is the analytic workspace created by Example 4–1.

```
List objectsToExport = new ArrayList();
objectsToExport.add(aw);
exportToXML(objectsToExport, "globalawj.xml");
```

***Example 4–9 Exporting to an XML Template***

```
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 4–10 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 4–10 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
- Kinds of Source Objects
- Characteristics of Source Objects
- Inputs and Outputs of a Source
- Describing Parameterized Source Objects
- Model Objects and Source Objects

## 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 the "Inputs and Outputs of a Source" topic. Some Source objects have an associated Model object, which is discussed in the "Model Objects and Source Objects" topic.

## Data Type of a Source

As described in Chapter 2, "Understanding OLAP Java API Metadata", the OLAP Java API has a class, FundamentalMetadataObject, that 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".

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.

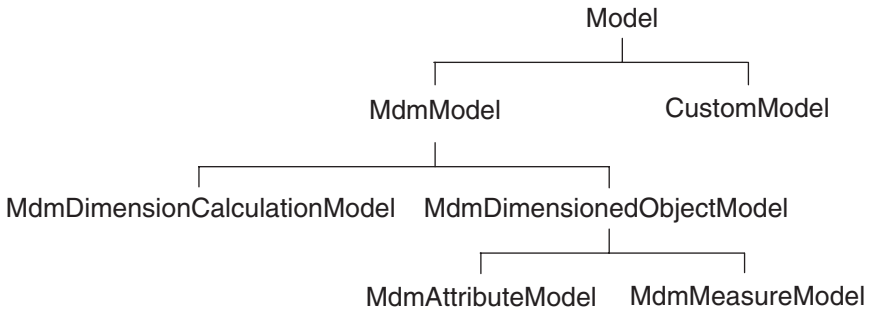
The value that the `Assignment` assigns is specified by a `Source`. An `Assignment` also has an integer that specifies a precedence that affects the order in which Oracle OLAP calculates a value and assigns it. If you create more that one `Assignment` for a `Model` without specifying a precedence, then 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

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 Query in a Transaction
- Using TransactionProvider Objects

## About Creating 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 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 the 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", 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***

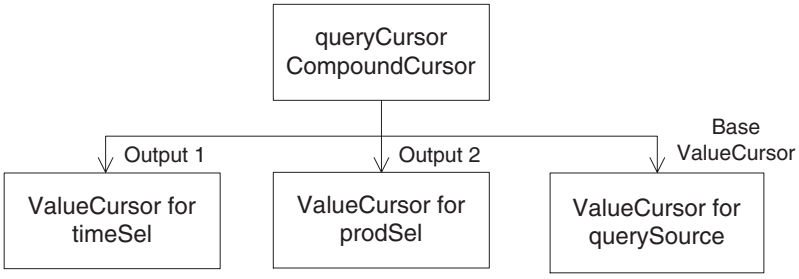
```
Source timeSel = timeHier.selectValues(new String[]
                                        {"CALENDAR_YEAR::MONTH::2001.01",
                                         "CALENDAR_YEAR::MONTH::2001.04",
                                         "CALENDAR_YEAR::MONTH::2001.07",
                                         "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.

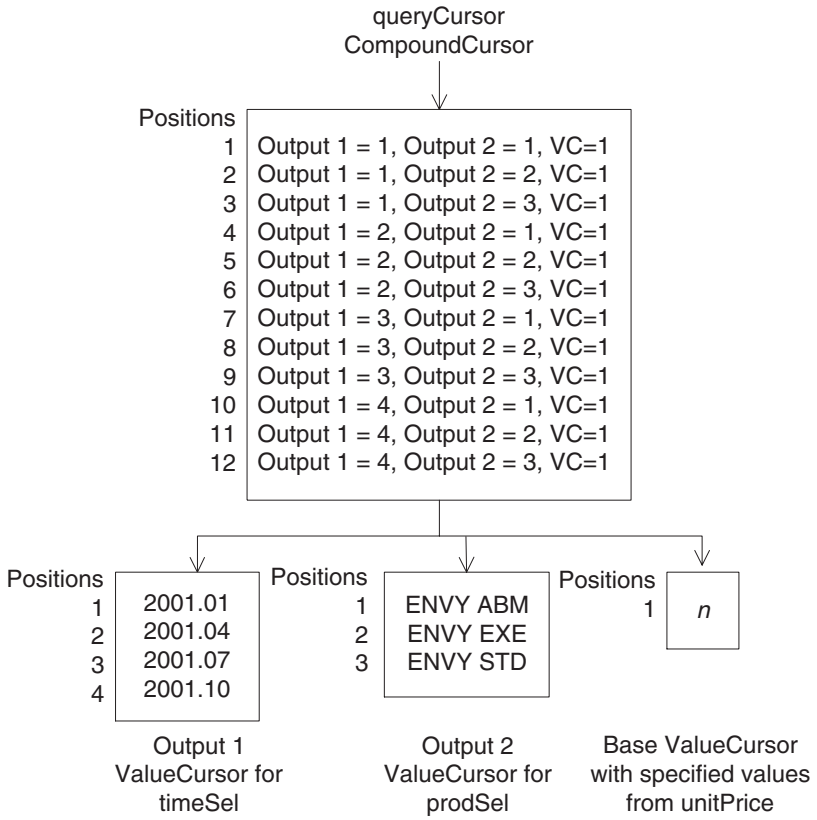
The result set for `querySource` has one measure value for each tuple (each set of output values), so the total number of values is twelve (one value for each of the three products for each of the four months). Therefore, the `queryCursor CompoundCursor` created for `querySource` has twelve positions.

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.

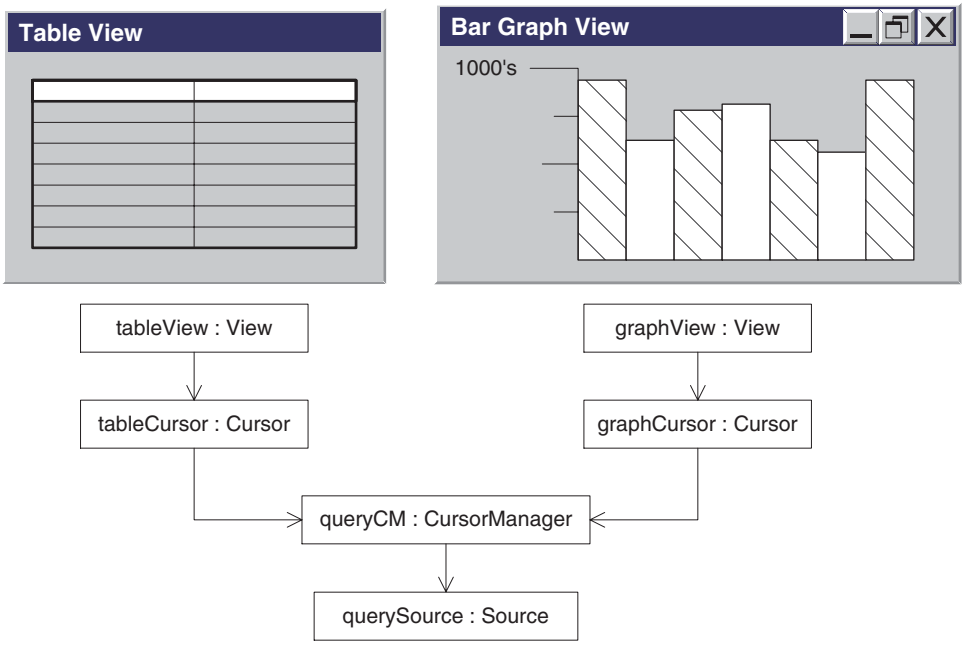
You can also regulate the number of elements that Oracle OLAP returns by using `Parameter` and parameterized `Source` objects in constructing your query. For more information on `Parameter` objects, see Chapter 5, "Understanding Source Objects". For examples of using parameterized `Source` objects, see 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
- Navigating a CompoundCursor for Different Displays of Data
- Specifying the Behavior of a Cursor
- Calculating Extent and Starting and Ending Positions of a Value
- Specifying a Fetch Size

## 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

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 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 on page 10-5.

## 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 on page 10-8.

## 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 on page 10-8.

### 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 on page 10-10.

## 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;
    if (castState.topBottomType == TOP_BOTTOM_TYPE_TOP)
      sortedBase = castState.base.sortDescending(castState.criterion);
    else
      sortedBase = castState.base.sortAscending(castState.criterion);
    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 from Chapter 7, 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
   * MdmMeasure objects for the UNITS and SALES.
   * 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.

// Get the short value descriptions of the dimension members from the
// SingleSelectionTemplate.
StringBuffer shortDescrsForMemberVals =
                           singleSelections.getMemberShortDescrs(dp);
println("\nThe " + Math.round(topNBottom.getN()) +
                " products with the most units sold \nfor" +
                shortDescrsForMemberVals + " are:\n");

// Get the Source produced by the TopBottomTemplate, create a Cursor
// for it, and display the values of the Cursor.
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 ContextExample11g.

// 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);
```

```
  // After changing the selection of a dimension member, get the short value
  // descriptions of the dimension members again.
  StringBuffer shortDescrsForMemberValsAfter =
                              singleSelections.getMemberShortDescrs(dp);

  // 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();

  println("\nThe " + Math.round(topNBottom.getN()) + " products " +
               "with the fewest units sold \nfor" +
                 shortDescrsForMemberValsAfter + " are:\n");

  // 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);

  println("\nThe " + Math.round(topNBottom.getN()) +
          " products with the highest sales amounts \nfor" +
          shortDescrsForMemberVals +" are:\n");

  topNBottomResult = topNBottom.getSource();
  result = prodShortDescrAttr.joinHidden(topNBottomResult);

  // Commit the current transaction.
  getContext().commit();

  // Change the type of selection back to the bottom.
  topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);

  println("\nThe " + Math.round(topNBottom.getN()) +
          " products with the lowest sales amounts \nfor" +
          shortDescrsForMemberVals +" are:\n");

  topNBottomResult = topNBottom.getSource();
  result = prodShortDescrAttr.joinHidden(topNBottomResult);

  // Commit the current transaction.
  getContext().commit();

}
```

```
  /**
   * 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. O/S Documentation Set - Italian
 2. External 48X CD-ROM
 3. O/S Documentation Set - Spanish
 4. Internal 48X CD-ROM USB
 5. Monitor- 19"Super VGA

The 7 products with the highest sales amounts
for Asia Pacific, Direct Sales, 2001 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, 2001 are:

 1. Keyboard Wrist Rest
 2. Mouse Pad
 3. O/S Documentation Set - Italian
 4. O/S Documentation Set - Spanish
 5. Standard Mouse
 6. O/S Documentation Set - French
 7. Internal 48X CD-ROM USB
```

# 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 Oracle Database installation, with the OLAP option, provides the OLAP Java API and other class libraries, as `jar` files, that you require to develop an OLAP Java API client application. 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.

## 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 installation does not provide the JDK. For information about obtaining and using it, see the Sun Microsystems Java Web site at

  `http://java.sun.com`

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.

# 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` |

The `ORACLE_HOME/olap/api/lib` directory also contains the `olap_api_doc.jar` file, which contains the *Oracle OLAP Java API Reference* documentation. You can add the JAR file to your IDE to get context-sensitive help for the API.

# 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

## A



## B



## C

## D



## E

## F



## G



## H



## I



## J



## L

## M

## N



## O



## P



## Q



## R

## S



## T

## U



## V



## W



## X