Oracle® OLAP

Developer's Guide to the OLAP API

10*g* Release 1 (10.1)

Part No. B10335-02

December 2003

ORACLE®

Oracle OLAP Developer's Guide to the OLAP API, 10*g* Release 1 (10.1)

Part No. B10335-02

Contents

2 Understanding OLAP API Metadata

3 Connecting to a Data Store

4 Discovering the Available Metadata

5 Working with Metadata Mapping Objects

6 Understanding Source Objects

7 Making Queries Using Source Methods

8 Using a TransactionProvider

9 Understanding Cursor Classes and Concepts

10 Retrieving Query Results

11 Creating Dynamic Queries

A Setting Up the Development Environment

Index

Send Us Your Comments

Oracle OLAP Developer's Guide to the OLAP API, 10*g* Release 1 (10.1)

Part No. B10335-02

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, then please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: 781-238-9850 Attn: Oracle OLAP
- Postal service:
  Oracle Corporation
  Oracle OLAP Documentation
  10 Van de Graaff Drive
  Burlington, MA 01803
  U.S.A.

If you would like a reply, please give your name, address, telephone number, and electronic mail address (optional).

If you have problems with the software, please contact your local Oracle Support Services.

Preface

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

The preface contains these topics:

Audience

This manual is intended for Java programmers who are responsible for creating applications that 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. 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 additional information, visit the Oracle Accessibility Program Web site at

http://www.oracle.com/accessibility/

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, 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, JAWS 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.

Structure

The following paragraphs describe the chapters that comprise this manual.

Chapter 1, "Introduction to the OLAP API"

Introduces the OLAP API to application developers who plan to use it in their Java applications.

Chapter 2, "Understanding OLAP API Metadata"

Describes the multidimensional metadata (MDM) classes that the OLAP API provides, and explains how MDM objects relate to the metadata objects that a database administrator specifies when preparing the OLAP Catalog.

Chapter 3, "Connecting to a Data Store"

Explains the procedure for connecting to a data store through the OLAP API.

Chapter 4, "Discovering the Available Metadata"

Explains the procedure for discovering the MDM metadata in a data store through the OLAP API.

Chapter 5, "Working with Metadata Mapping Objects"

Describes the metadata mapping (MTM) classes that the OLAP API provides, and explains how MTM objects relate to the mapping of MDM metadata objects to relational database tables or views.

Chapter 6, "Understanding Source Objects"

Introduces `Source` objects, which are the OLAP API objects that specify a query of the data in the database.

Chapter 7, "Making Queries Using Source Methods"

Provides examples of using the basic `Source` methods and of using `Source` methods to accomplish typical OLAP tasks and other operations.

Chapter 8, "Using a TransactionProvider"

Describes the Oracle OLAP API `Transaction` and `TransactionProvider` interfaces and describes how you use implementations of those interfaces in an application. You must create a `TransactionProvider` before you can create a `DataProvider`, and you must use methods of the `TransactionProvider` to prepare and commit a `Transaction` before you can create a `Cursor` for a derived `Source`.

Chapter 9, "Understanding Cursor Classes and Concepts"

Describes the Oracle OLAP API `Cursor` class and its related classes, which you use to retrieve and gain access to the results of a query. This chapter also describes the `Cursor` concepts of position, fetch size, and extent.

Chapter 10, "Retrieving Query Results"

Describes how to retrieve the results of a query with an Oracle OLAP API `Cursor`, how to gain access to those results, and how to customize the behavior of a `Cursor` to fit your method of displaying the results.

Chapter 11, "Creating Dynamic Queries"

Describes the Oracle OLAP API `Template` class and its related classes, which you use to create dynamic queries. This chapter also provides examples of implementations of those classes.

Appendix A, "Setting Up the Development Environment"

Describes the steps you take to set up your development environment for creating applications that use the OLAP API.

Related Documents

For more information, see the following manuals in the Oracle 10*g* Release 1 (10.1) documentation set:

- *Oracle OLAP Java API Reference*
- *Oracle OLAP Application Developer's Guide*
- *Oracle OLAP Analytic Workspace Java API Reference*
- *Oracle OLAP Reference*
- *Oracle OLAP DML Reference*
- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Data Warehousing Guide*

Conventions

The following conventions are also used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted. In an example, they mean that information not directly related to the example has been omitted.
boldface text	Boldface type in text indicates a term defined in the text.
italic text	Italic typeface denotes book titles or emphasis.
`monospace text`	Monospace typeface indicates filenames, Java elements, such as classes, methods, and fields, SQL keywords or commands, code examples, and references to code example objects in the text. An exception is the OLAP API data types, which appear in regular typeface to distinguish them from Java data types.
`monospace italic text`	Monospace italic typeface indicates a variable that a user supplies in a SQL command.

1

Introduction to the OLAP API

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

This chapter includes the following topics:

OLAP API Overview

The OLAP API is a Java application programming interface (API) through which an application can access data for online analytical processing (OLAP). The Java classes that implement the API are part of the Oracle OLAP component.

The purpose of the OLAP API is to facilitate the development of OLAP applications, which allow users 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 API are optimized for this type of application. With the OLAP API, a Java application can access, manipulate, and display data in multidimensional terms. The OLAP 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 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 API:

- Dimension. A structure that categorizes data. Commonly used dimensions are customers, products, and times. Typically, a dimension is associated with one or more hierarchies. Several distinct dimensions, combined with measures, enable end users to answer business questions. For example, a times dimension that categorizes data by month helps to answer the question, "Did we sell more widgets in January or June?"
- Measure. Data, usually numeric and additive, that can be examined and analyzed. Typically, a given measure is categorized by one or more dimensions, and it is described as "dimensioned by" them.
- Hierarchy. A logical structure that uses ordered levels or values as a means of organizing dimension elements in parent-child relationships. Typically, end users can expand or collapse the hierarchy by drilling down or up on its levels.
- Level. A position in a level-based hierarchy. For example, a times dimension might have a hierarchy that represents data at the day, month, quarter, and year levels.
- Attribute. A descriptive characteristic of the elements of a dimension that an end user can specify to select data. For example, end users might choose products using a color attribute.
- Query. A specification for a particular set of data, and for aggregations, calculations, or other operations to perform using the data. Any such operations on the data are an intrinsic part of the query. The data and the operations on it define the result set of the query.

Two additional data warehouse and OLAP concepts, cube and edge, are not intrinsic to the OLAP API, but are often incorporated into the design of applications that use the OLAP API.

- Cube. A logical organization of multidimensional data. Typically, the edges of a cube contain dimension 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 from the time, product, customer, and channel dimensions and whose body contains values from the units sold measure.

- Edge. One side of a cube. Each edge contains values 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 the *Oracle Data Warehousing Guide*.

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

The OLAP 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.

In order for the OLAP API to access the data in a given data warehouse, a database administrator must first 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 must map the data to OLAP metadata objects and add them to the OLAP Catalog. Finally, with the metadata in place, an application can access both the data and the metadata through the OLAP API.

See the *Oracle OLAP Application Developer's Guide* for information about supported data warehouse configurations and about creating OLAP Catalog metadata.

The collection of warehouse data for which a database administrator has mapped to OLAP Catalog elements is the data store to which the OLAP API gives access. Of course, each user who accesses data through the OLAP API might have security restrictions that limit the scope of the data that he or she can access within the data store.

With the classes in the `oracle.olapi.metadata.mtm` package, an application developer who is familiar with SQL and with the mapping of the relational tables and views to the OLAP Catalog metadata can create custom metadata objects. For more information, see Chapter 5, "Working with Metadata Mapping Objects".

What Can an Application Do with the OLAP API?

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

- Establish a connection to a data store.
- 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 multidimensional format.
- Modify existing queries, rather than totally redefine them, as application users refine their analyses.

Context for OLAP API Development

The OLAP API is a Java API, so it 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 API, and because the client application is written in Java, its code can also take advantage of them.

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

Sample Schema for OLAP API Examples

This documentation has examples of OLAP API code that use a relational schema, named the Global schema, and an analytic workspace generated from that relational schema. For the complete code of the examples in this documentation, see the Overview of the *Oracle OLAP Java API Reference*.

The OLAP Catalog for the Global schema has the following measures:

- `UNITS`, which has the quantities of product units sold.
- `UNIT_COST`, which has the cost of a unit.
- `UNIT_PRICE`, which has the price of a unit.

The data in the measures is identified by detailed (leaf-level) data or aggregate (node-level) data from dimensions. The `UNIT` measure is dimensioned by the following dimensions:

- PRODUCT, which has a hierarchy of product values named PRODUCT_ROLLUP. The leaf level of the hierarchy has product item identification numbers and the higher levels have product family, class, and total products identifiers.
- CUSTOMER, which has two hierarchies of customer values, named SHIPMENTS_ROLLUP and MARKET_ROLLUP. The lowest level of each hierarchy has customer identification numbers and higher levels have warehouse, regions, and total customers, and accounts, market segments, and total market identifiers, respectively.
- TIME, which has a hierarchy of calendar year time period identifiers.
- CHANNEL, which has a hierarchy of sales channel identifiers.

The UNIT_COST and UNIT_PRICE measures are dimensioned by the following two dimensions:

- PRODUCT
- TIME

For an example of a program that discovers the OLAP Catalog metadata for the Global schema, see Chapter 4, "Discovering the Available Metadata".

Access to Data and Metadata Through the OLAP API

The OLAP Catalog metadata describes the data that is available to the OLAP API through a connection to the database. The metadata records three things:

- The existence of sets of data. For example, a measure of unit price figures, dimensions of product and time values, and attributes that contain information about the elements 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 elements 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 element values, the dimensions have String values that identify the product or time values and the hierarchical levels, and the dimensions have attributes that provide additional information, such as a descriptive name for each dimension element that can be used in reports.

In contrast, the fact that the price of product 13, which is the Envoy Standard portable PC, was 2426.07 dollars in July 1002 is data, not metadata.

These examples distinguish between the metadata and the data for the measure of unit prices. The OLAP API makes a similar distinction between the metadata and the data for dimensions. For example, the fact that a product dimension exists and that it has text values as elements is metadata. In contrast, the fact that the value of one of its elements is 13 is data.

MDM Model in the OLAP API

The OLAP 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 measures, dimensions, hierarchies, and attributes.

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

- `MdmSchema`
- `MdmMetadataProvider`
- `MdmMeasure`
- `MdmDimension`
- `MdmHierarchy`
- `MdmLevel`
- `MdmAttribute`

An `MdmSchema` is a container for `MdmMeasure`, `MdmDimension`, and other `MdmSchema` objects. An `MdmSchema` corresponds to a measure folder in the OLAP management feature of Oracle Enterprise Manager. Note that an `MdmSchema` does not necessarily correspond to a relational schema.

An `MdmMetadataProvider` gives an application access to metadata objects that were created by a database administrator using the OLAP management feature of Oracle Enterprise Manager. To obtain access to the metadata, an application uses the `getRootSchema` method of an `MdmMetadataProvider`. This method returns the top-level `MdmSchema`, which contains all of the `MdmDimension` objects that are accessible through this particular `MdmMetadataProvider`. The `MdmDimension` objects might be organized in a hierarchical tree, with subschemas nested under the top-level schema. Using the `getMeasureDimension`, `getSubSchemas`, and `getDimensions` methods of the top-level `MdmSchema`, and the `getSubSchemas`,

getMeasures, and getDimensions methods of all of the nested MdmSchema objects, an application navigates through the metadata and discovers what data is available. In addition, the application can use methods to obtain the related MdmMeasure, MdmHierarchy, MdmLevel, and MdmAttribute objects.

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

Access to Data Through the OLAP API

An MdmMeasure or MdmDimension represents data in the data store. For example, an MdmMeasure object named units might represent a set of numeric elements whose values are dollar amounts for units sold, and an MdmDimension called prodDim might represent a set of text elements whose 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 calls the getSource method of the MdmMeasure or MdmDimension. This method creates a Source object that specifies 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. Each new query is a new Source.

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 API.

Because the primary focus of most OLAP applications is making queries against the data store, a significant proportion of their data manipulation code works with the

following classes, each of which has methods for selecting, calculating, and otherwise manipulating data.

- `Source`
- `BooleanSource`
- `DateSource`
- `NumberSource`
- `StringSource`

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.

Unique and Local Dimension Values

The elements of an OLAP Catalog 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. In the OLAP API a dimension always has at least one hierarchy dimension object and that hierarchy object has at least one level object. Even a nonhierarchical dimension is represented by a hierarchy dimension object with one level object.

The OLAP API uses a three-part format to specify the hierarchy, the level, and the value of a dimension element, 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 element 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 value in the `YEAR` level of the `CALENDAR` hierarchy of the `TIME` dimension:

```
CALENDAR::YEAR::2
```

The third part of a unique value is the local value. The local value in the preceding example identifies the year 1999.

The OLAP API has classes and methods that you can use to get the local values of dimension elements. The `MdmPrimaryDimension` class has a method for getting an `MdmAttribute` that records the local values for the elements of the hierarchies that are components of the `MdmPrimaryDimension`, and the `MdmDimensionMemberInfo` class has methods for getting the local or unique values for 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 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 Application Developer's Guide*.

OLAP API Client Software

The OLAP 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 discover metadata, specify queries, and retrieve data.

When a Java application calls methods of objects of OLAP API Java classes, it uses the OLAP API client software to communicate with Oracle OLAP, which resides within an Oracle database instance. The communication between the OLAP API client software and Oracle OLAP is provided through Java Database Connectivity (JDBC), which is a standard Java interface for connecting to relational databases. For more information about JDBC, see the *Oracle Database JDBC Developer's Guide and Reference*.

Requirements for Using the OLAP API Client Software

To use the OLAP API classes as you develop your application, import them into your Java code. When you deliver your application to users, include the OLAP API classes with the application. You must also ensure that users can access JDBC.

In order to develop an OLAP 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 you used for development.

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

Tasks That an OLAP API Application Performs

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

1. Connects to the data store
2. Discovers the available metadata
3. Specifies queries that select and manipulate data
4. 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 and specifying this information in a JDBC connection method.

For more information about connecting, see Chapter 3, "Connecting to a Data Store".

Task 2: Discover the Available Metadata

Having established a connection, the application creates an `MdmMetadataProvider`. This object 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 `MdmSchema` object that represents the top-level measure folder for all of the metadata objects to which the `MdmMetdataProvider` provides access. The application then gets the dimensions, including the measure dimension, and the subfolders that are under the root.

Once the application has all of the dimensions, it can interrogate them to get their attributes, hierarchies, levels, and other characteristics, and the measures. 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 API Metadata". For information about how an application can discover the available metadata, see Chapter 4, "Discovering the Available Metadata".

Task 3: Select and Calculate Data Through Queries

The heart of any OLAP application lies in the construction of queries against the data store. The application user interface provides ways for the user to select data and to specify what should be done with it. 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 elements, or they can be complex, including several aggregations and calculations on measure values specified by selections of dimension elements.

The OLAP API object that specifies a query is a `Source`. Therefore, a significant portion of any OLAP API application is devoted to dealing with `Source` objects.

From an `MdmSchema`, you get `MdmSource` objects, such as an `MdmMeasure` or an `MdmPrimaryDimension`. You then get a `Source` object from the `MdmSource`. 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, you might use only the methods of the `Source` classes 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, 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 minimize your work by writing general-purpose `Template` classes and reusing them in various parts of your application.

For information about working with `Source` objects, see Chapter 6, "Understanding Source Objects". For information about working with `Template` objects, see Chapter 11, "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 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 API. The ExpressSQLCursorManager class implements the SQLCursorManager interface.

Because the OLAP 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 structured by four MdmPrimaryDimension objects. Each MdmPrimaryDimension is represented by a 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 measure data as its values and it has the Source objects for the dimensions as its outputs.

A Cursor for the query Source has the same structure as the Source; that is, the values of the Cursor 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 6, "Understanding Source Objects". For more information about using Cursor objects to retrieve data, see Chapter 9, "Understanding Cursor Classes and Concepts". For more information about the SQLCursorManager class, see the *Oracle OLAP Java API Reference*.

2

Understanding OLAP API Metadata

This chapter describes the metadata objects that the OLAP API provides, and explains how these objects relate to the OLAP metadata objects that a database administrator specifies in the OLAP Catalog or that you create with the methods of an `MdmCustomObjectFactory` or an `MtmPartitionedCube`.

This chapter includes the following topics:

For the complete code of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

Overview of the OLAP API Metadata

The OLAP API provides a Java application with access to a multidimensional view of data in an Oracle database. The OLAP API design includes objects that are consistent with that view and are familiar to data warehousing and OLAP

developers. For example, it has objects for measures, dimensions, hierarchies, levels, and attributes. The OLAP API design incorporates an object-oriented model called MDM (multidimensional metadata).

The data in an Oracle database must be prepared by a database administrator in order to support the MDM model. An administrator must map the relational data to OLAP metadata by adding objects and characteristics to the OLAP Catalog.

Data Preparation

A database administrator starts with a data warehouse that is organized according to certain specifications. For example, it might conform to a star schema. The requirements are described in the *Oracle OLAP Application Developer's Guide*.

Metadata Preparation

The administrator adds OLAP metadata to the OLAP Catalog. The OLAP metadata objects, which are created in this step, supply the metadata required for Oracle OLAP to access the data. These OLAP metadata objects map to MDM metadata objects in the OLAP API.

An application developer can discover the mapping of the MDM metadata objects to the relational tables and views, or create some custom MDM metadata objects, by using MTM (metadata mapping) objects. See Chapter 5, "Working with Metadata Mapping Objects", for more information on MTM objects.

The topic "OLAP Metadata Objects" briefly describes the OLAP metadata objects that a database administrator prepares for use with Oracle OLAP.

OLAP Metadata Objects

A database administrator adds OLAP metadata to the OLAP Catalog for a data warehouse. The end result is the creation of one or more measure folders that contain one or more measures. The measures have dimensions and the dimensions have hierarchies, levels, and attributes. Each of these OLAP metadata objects maps directly to an MDM object in the OLAP API. For detailed information about OLAP metadata and about adding metadata to the OLAP Catalog, see the *Oracle OLAP Application Developer's Guide*.

An application developer can create transient custom metadata objects with instances of an `MdmCustomObjectFactory` or an `MtmPartitionedCube` class. The developer uses methods of MTM objects to map the MDM objects to data in the

columns in relational tables. The transient objects exist only in the context of an `MdmMetadataProvider` during a connection to the database.

Note that the OLAP metadata includes a cube object, which does not map directly to any MDM object. Database administrators create cubes in the OLAP Catalog to specify the dimensions of each measure. Once the dimensions are specified, they are firmly associated with their measures in the metadata, so this type of cube object is not needed in the MDM model.

The rest of this topic briefly describes the OLAP metadata objects that map directly to MDM objects in the OLAP API.

Dimensions in the OLAP Metadata

The following are some of the characteristics that a database administrator can specify for dimensions:

- General characteristics, such as the name of the dimension and the database schema from which its data is drawn.
- Hierarchies, which organize the elements of the dimension into parent-child relationships. A hierarchy can be level-based or value-based. In a level-based hierarchy, the parent and child elements are in different levels. In a value-based hierarchy, the database administrator has defined the parent and child relationships by values rather than levels. A simple, nonhierarchical list of elements is represented by a hierarchy that has only one level and that has no parent-child relationships defined for the elements.
- Levels, which organize the elements of a hierarchy into the levels defined for the dimension.
- Attributes, which record characteristics of the elements for the dimension. For example, attributes record the level of each element of a level-based hierarchy and the depth of that level in the hierarchy.

Typically, a database administrator specifies one or more columns in a database table to serve as the basis for each OLAP level, hierarchy, and attribute.

A database administrator creates cubes after creating dimensions. A cube is a set of dimensions that provide organizational structure for measures.

Measures in the OLAP Metadata

In the OLAP Catalog, a database administrator specifies that a given measure belongs to a given cube. Because a cube is a set of dimensions that provide

organizational structure for measures, specifying that a given measure belongs to a given cube specifies the dimensions of that measure. This is essential information for the OLAP API, where the dimensionality of a measure is one of its most important features.

To identify the data for a measure, the database administrator typically specifies a column in a fact table where the data for the measure resides. As an alternative, the database administrator can specify a calculation or transformation that produces the data.

Measure Folders in the OLAP Metadata

Once a database administrator has created measures (after first creating dimensions and cubes), the next step is to create one or more groups of measures called measure folders. Typically, the measures in a given folder are related by subject matter. That is, they all pertain to the same business area. For example, there might be separate folders for financials, sales, and human resources data.

The measures in a given measure folder can belong to different cubes and they can be from more than one schema.

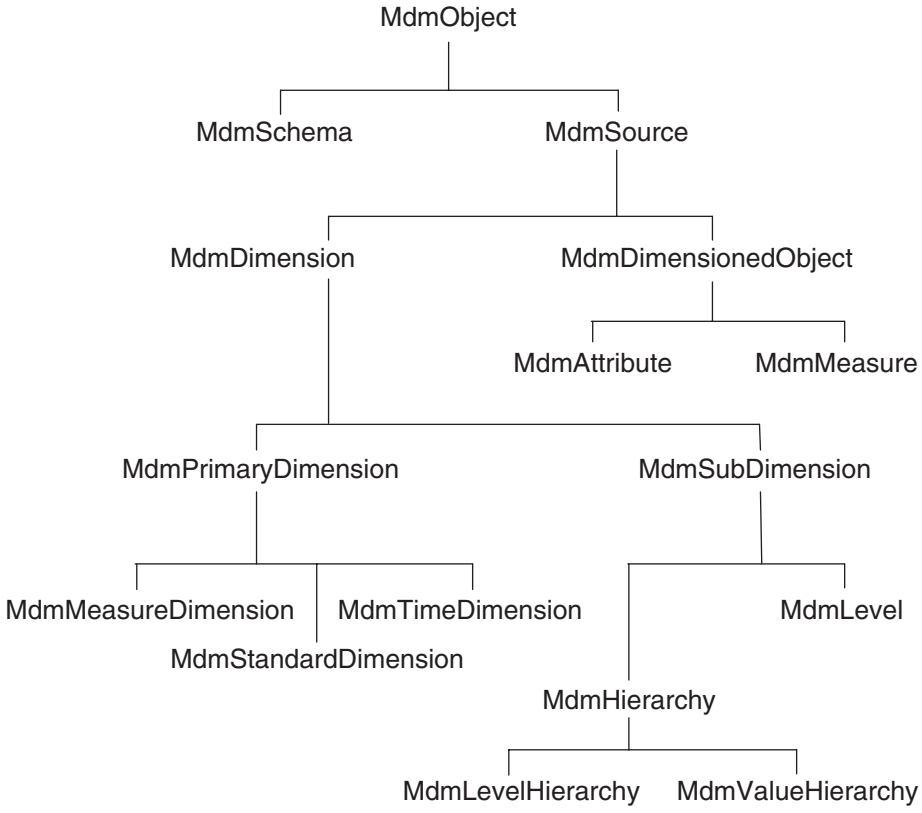
The database administrator must create at least one measure folder because the scope of the data that an OLAP API application can access is defined in terms of measure folders. That is, an OLAP API `MdmMetadataProvider` gives access only to the measures that are contained in measure folders. Of course, the dimensions of each measure are included, along with their hierarchies, levels, and attributes.

In this context, it is important to understand that measure folders can be nested. This means that a given measure folder can have subfolders that have their own measures, and even their own subfolders. Thus, a database administrator can arrange measures in a hierarchy of folders, and an OLAP API `MdmMetadataProvider` can give access to all of the measure folders and their subfolders.

Overview of MDM Metadata Objects in the OLAP API

The OLAP API implementation of the MDM model is represented by classes in the `oracle.olapi.metadata.mdm` package. Most of the classes in this package implement metadata objects, such as dimensions and measures. The following diagram introduces the subclasses of the `MdmObject` class.

Figure 2–1 MdmObject Class and Its Subclasses



Mapping of OLAP Metadata Objects to MDM objects

An application gains access to metadata objects by creating an OLAP API `MdmMetadataProvider` and using it to discover the available metadata objects in the data store.

The metadata objects that a database administrator specifies in the OLAP Catalog map directly to MDM metadata objects that are accessible through the `MdmMetadataProvider`. The following table presents a typical mapping.

OLAP Catalog Metadata Objects	MDM Metadata Objects
Dimension	`MdmPrimaryDimension`
Hierarchy	`MdmLevelHierarchy` or `MdmValueHierarchy`
Level	`MdmLevel`
Measure	`MdmMeasure`
Attribute	`MdmAttribute`
Measure folder	`MdmSchema`

This chapter describes the MDM metadata objects. For information about how an application discovers the available MDM metadata objects in the data store, see Chapter 4, "Discovering the Available Metadata". MTM objects record the mapping of MDM objects to relational tables. For information on MTM objects, see Chapter 5, "Working with Metadata Mapping Objects".

`MdmSchema` and `MdmSource` are the two subclasses of `MdmObject`.

MdmSchema Class

An `MdmSchema` represents a set of data that is used for navigational purposes. An `MdmSchema` is a container for `MdmMeasure`, `MdmPrimaryDimension`, and other `MdmSchema` objects. An `MdmSchema` 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 a measure folder, which can include data from several relational schemas and which was created in the OLAP Catalog by a database administrator or by the `createSchema` method of an `MdmCustomObjectFactory`.

Data that is accessible through the OLAP API is arranged under a top-level `MdmSchema`, which is referred to as the root `MdmSchema`. Under the root, there are one or more subschemas. To begin navigating the metadata, an application calls the `getRootSchema` method of the `MdmMetadataProvider`, as explained in Chapter 4, "Discovering the Available Metadata".

The root `MdmSchema` contains all of the `MdmDimension` objects that are in the data store. Most `MdmPrimaryDimension` objects are contained in subschemas under the root `MdmSchema`. However, a data store can contain a dimension that is not included in a subschema. The root `MdmSchema` contains dimension objects that are in subschemas as well as dimension objects that are not.

The root MdmSchema contains MdmMeasure objects only if they are not contained in a subschema. Because most MdmMeasure objects belong to a subschema, the root MdmSchema typically has no MdmMeasure objects. Therefore, the getMeasures method of the root MdmSchema typically returns an empty List object.

An MdmSchema has methods for getting all of the MdmMeasure, MdmPrimaryDimension, and MdmSchema objects that it contains. The root MdmSchema also has a method for getting the MdmMeasureDimension, whose elements are all of the MdmMeasure objects in the data store regardless of whether they belong to a subschema.

MdmSource Class

An MdmSource represents a measure, dimension, or other set of data (such as an attribute) that is used for analysis. MdmSource objects represent data, but they do not provide the ability to create queries on that data. Their function is informational, recording the existence, structure, and characteristics of the data. They do not give access to the data values.

To gain access to the data values for a given MdmSource, an application calls the getSource method of the MdmSource. This method returns a Source through which an application can create queries on the data represented by the MdmSource. The following line of code creates a Source from 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. Each new Source specifies a new query.

For more information about working with Source objects, see Chapter 6, "Understanding Source Objects". The rest of this chapter describes the subclasses of MdmSource.

MdmDimension Class

MdmDimension is an subclass of MdmSource. The abstract MdmDimension class represents the general concept of a list of elements 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 element of the MdmDimension to a value representing some characteristic of the element value. To obtain the MdmAttribute objects for an MdmDimension, call its getAttributes method.

MdmDimension has the abstract subclasses MdmPrimaryDimension and MdmSubDimension.

MdmPrimaryDimension Class

MdmPrimaryDimension is an abstract subclass of MdmDimension. An MdmPrimaryDimension represents the dimensions defined by the database administrator in the OLAP Catalog. The concrete subclasses of the abstract 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 its elements. A data store has only one MdmMeasureDimension. You can obtain the MdmMeasureDimension by calling the getMeasureDimension method of the root MdmSchema and casting the result to an MdmMeasureDimension. 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 its elements. 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 MdmHierarchy objects, which represent the hierarchies of the dimension. An MdmPrimaryDimension has all of the elements of its component MdmHierarchy objects, while each of its MdmHierarchy objects has only the elements in that hierarchy.

An MdmPrimaryDimension that represents a nonhierarchical list of elements has only one MdmLevelHierarchy, which has all of its elements at one level with no hierarchical relationships defined for them. For example, the

MdmMeasureDimension represents a dimension that is simple list of the MdmMeasure objects in the data store. The MdmMeasureDimension has one MdmLevelHierarchy, which has one MdmLevel. The MdmMeasureDimension, its MdmLevelHierarchy, and its MdmLevel all have the same dimension elements, the values of which are the MdmMeasure objects.

MdmHierarchy Class

MdmHierarchy is an abstract subclass of MdmSubDimension, which is an abstract subclass of MdmDimension. An MdmHierarchy represents an organization of the elements of an MdmPrimaryDimension, which can have more than one hierarchy defined for it. 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 elements of both hierarchies are drawn from the elements of the MdmTimeDimension, but the number of elements in each hierarchy and the parent-child relationships of the values of the elements can be different.

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.

MdmLevelHierarchy

MdmLevelHierarchy is a concrete subclass of MdmHierarchy. An MdmLevelHierarchy has its parent-child relationships defined between the values of the elements at different levels. The different levels of an MdmLevelHierarchy are represented by MdmLevel objects. An MdmLevelHierarchy has a tree-like structure. The elements at the lowest level of the hierarchy are the leaves, and the elements at higher levels are nodes. Nodes have children; leaves do not.

The MdmLevelHierarchy has all of the elements of the hierarchy, and each of its component MdmLevel objects has only the elements at the level it represents. Each element, except those at the highest level, can have a parent, and each element, except those at the lowest level, can have one or more children. The parent and children of an element of an MdmLevel are in other MdmLevel objects. An MdmLevelHierarchy can also represent a nonhierarchical list of elements, in which case the MdmLevelHierarchy has one MdmLevel, and both objects have the same elements. You get the levels of an MdmLevelHierarchy by calling its getLevels method.

MdmValueHierarchy

MdmValueHierarchy is the other concrete subclass of MdmHierarchy. An MdmValueHierarchy has parent-child relationships defined between the values of the dimension elements, and does not have the parent and child elements at different levels. 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. A database administrator defines a dimension as a value hierarchy in the OLAP Catalog. An application developer can define a value hierarchy with the createValueHierarchy method of an MdmCustomObjectFactory.

MdmLevel Class

MdmLevel is a subclass of MdmSubDimension. An MdmLevel represents a list of elements that supply one level of the hierarchical structure of an MdmLevelHierarchy.

An MdmLevel represents a level that was specified by a database administrator in the OLAP Catalog. Typically, a database administrator specifies a column in a relational database table to provide the values of the level. The values of the elements of an MdmLevel must be unique. If the column in the database has values that are not unique, then the database administrator can define the elements of a level using two or more columns of the table, thus ensuring that the elements of the MdmLevel have unique values. For example, if a dimension of geographical locations has a level for cities and more than one city has the same name, then a database administrator can specify as the value of the city level both the city column and the state column in the relational database. The values of the elements in the MdmLevel for cities are then combinations of the two column values, such as IL:Springfield for Springfield, Illinois and MA:Springfield for Springfield, Massachusetts.

An MdmLevelHierarchy has one MdmLevel for each level of elements in the hierarchy of dimension elements that it represents. Each element of an MdmLevel, except the highest level, can have a parent, and each element, except those of the lowest level, can have one or more children. The parent and children of elements of one MdmLevel are elements from other MdmLevel objects.

Even though the elements of an MdmLevel have parent-child relationships, an MdmLevel is represented as a simple list. The parent-child relationships among the elements are recorded in the parent and ancestors attributes, which you can obtain by calling the getParentAttribute and getAncestorsAttribute methods of the MdmLevelHierarchy of which the MdmLevel is a component. You can get the

MdmLevelHierarchy for the MdmLevel by calling the getLevelHierarchy method of the MdmLevel.

MdmMeasure Class

MdmMeasure is a subclass of MdmDimensionedObject, which is an abstract subclass of MdmSource.

Description of an MdmMeasure

An MdmMeasure 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 for product units sold data, and the data 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 products, times, customers, and channels dimensions providing the organizational structure. The values of these four dimensions are indexes for identifying each particular cell in the array, which contains a single units sold data value. You must specify a value for each dimension in order to identify a value in the array. In relational terms, the MdmDimension objects constitute a compound (that is, composite) primary key for the MdmMeasure.

The values of an MdmMeasure are usually numeric, but this is not necessary.

Elements of an MdmMeasure

An MdmMeasure is based on an OLAP measure that was created by a database administrator in the OLAP Catalog or is created by an application developer using a method of an MdmCustomObjectFactory. In most cases, the MdmMeasure maps to a column in a fact table or to an expression that specifies a mathematical calculation or a data transformation. In many but not all cases, the MdmMeasure also maps to at least one hierarchy for each OLAP dimension of the measure, as well as an aggregation method. Oracle OLAP uses all of this information to identify the number of elements in the MdmMeasure and the value of each element.

MdmMeasure Elements Are Determined by MdmDimension Elements

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

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 elements of their component MdmHierarchy objects. Because of this structure, the values of the elements of an MdmMeasure are of two kinds.

- Values from the fact table column (or fact-table calculation) on which the MdmMeasure is based, as specified in the OLAP Catalog or the custom object. These values belong to MdmMeasure elements that are identified by a combination of values from the elements 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 element from a node level of an MdmHierarchy.

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 elements and node elements of the dimension it represents.

A unique combination of two elements, 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 elements (for example, a particular product item and a particular month). Other mdmUnitCost elements are identified by a combination of node elements (for example, a particular product family and a particular quarter). Still other mdmUnitCost elements are identified by a mixture of leaf and node elements. The values of the mdmUnitCost elements that are identified only by leaf elements 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 element, 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.

MdmAttribute Class

MdmAttribute is a subclass of MdmDimensionedObject, which is an abstract subclass of MdmSource.

Description of an MdmAttribute

An MdmAttribute represents a particular characteristic of the elements of an MdmDimension. An MdmAttribute maps one element of the MdmDimension to a particular value.

For example, mdmCustDim is an MdmPrimaryDimension for the CUSTOMER dimension, which represents a dimension of customers and is based on the columns of the GLOBAL.CUSTOMER_DIM table of a relational database. The MdmPrimaryDimension has a hierarchy that has levels that are based on shipment origination and destination values. One column in the customer table records a short value description for the customer identification value. The MdmAttribute returned by the getShortValueDescriptionAttribute method of mdmCustDim relates a short description to each the element of the dimension. The elements of the MdmAttribute have String values such as Computer Services Tokyo, Italy, and North America.

The values of an MdmAttribute might be String values (such as Italy), numeric values (such as 45), or objects (such as MdmLevel objects).

Like an MdmMeasure, an MdmAttribute has elements that are organized by its MdmDimension. Sometimes an MdmAttribute does not have a value for every element of its 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_ROLLUP hierarchy of the CUSTOMER dimension, because contact information does not apply to the higher REGION and ALL_CUSTOMERS levels. If an MdmAttribute does not apply to an element of an MdmDimension, then the MdmAttribute element value for that element is null.

Some MdmAttribute objects provide a mapping that is one-to-many, rather than one-to-one. Therefore, a given element 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 element to its set of ancestor MdmHierarchy elements.

Some attributes apply generally to the all of elements of an MdmPrimaryDimension. You can get the MdmAttribute objects for those attributes with the getAttributes method of the MdmPrimaryDimension. Other attributes relate only to an MdmHierarchy, such as the parent attribute, or an

MdmLevelHierarchy, such as the level depth attribute. You get the MdmAttribute objects for those attributes with the getParentAttribute method of the MdmHierarchy or the getLevelDepthAttribute method of the MdmLevelHierarchy.

Elements of an MdmAttribute

An MdmAttribute is based on an attribute that was specified for a dimension, hierarchy, or level by a database administrator in the OLAP Catalog or that was specified by an MtmValueExpression for a custom MdmAttribute created by an application.

The following table lists the values of elements of a Source object that represents the elements 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 element of the dimension has a related short description. The other is a custom attribute that relates a color to the values of elements at the ITEM level, which is the lowest level of the hierarchy. The values of the color MdmAttribute are null for the aggregate TOTAL_PRODUCT, CLASS, and FAMILY levels. In the table, null values appear as NA.

Product Values	Related Short Descriptions	Related Colors
PRODUCT_ROLLUP::TOTAL_PRODUCT::1	Total Product	NA
PRODUCT_ROLLUP::CLASS::2	Hardware	NA
PRODUCT_ROLLUP::FAMILY::4	Portable PCs	NA
PRODUCT_ROLLUP::ITEM::13	Envoy Standard	Black
PRODUCT_ROLLUP::ITEM::14	Envoy Executive	Black
PRODUCT_ROLLUP::ITEM::15	Envoy Ambassador	Black
PRODUCT_ROLLUP::FAMILY::5	Desktop PCs	NA
PRODUCT_ROLLUP::ITEM::16	Sentinel Standard	Beige
PRODUCT_ROLLUP::ITEM::17	Sentinel Financial	Beige
PRODUCT_ROLLUP::ITEM::18	Sentinel Multimedia	Beige
...	...	...

Data Type and Type of MDM Metadata Objects

All `MdmSource` objects have the following two basic characteristics:

- Data type
- 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 API implements the concept of data type through the `FundamentalMetadataObject` and `FundamentalMetadataProvider` classes. Every data type recognized by the OLAP 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 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 API data types appear in regular typeface, instead of monospace typeface, to distinguish them from `java.lang` data type classes.

OLAP 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`

OLAP API Data Type	Description of the `FundamentalMetadataObject`	Method of `FundamentalMetadataProvider`
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 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 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 API numeric data types: Double, Float, Integer, and Short	`getNumberDataType`
Value	Represents a general data type that includes any or all of the OLAP API data types.	`getValueDataType`
Empty	Represents no data, for example when an `MdmSource` has no elements at all defined for it.	`getEmptyDataType`
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 its elements 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 `MdmSource` objects, such as `MdmMeasure`, `MdmLevelHierarchy`, and `MdmLevel`. The typical data type of an `MdmMeasure` is one of the numeric data types; the typical data type of an `MdmLevelHierarchy` or `MdmLevel` is String.

Getting the Data Type of an MdmSource

If you have obtained an `MdmSource` from the data store, and you want to find the data type of its elements, then you call its `getDataType` method. This method returns a `FundamentalMetadataObject`.

To find the OLAP API data type that is represented by the returned `FundamentalMetadataObject`, you compare it to the `FundamentalMetadataObject` for each OLAP 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 the `MdmSource` that is passed in as a parameter. Note that this code creates a `FundamentalMetadataProvider` by calling a method of a `DataProvider`. Getting a `DataProvider` is described in Chapter 4, "Discovering the Available Metadata". The `dp` object is the `DataProvider`.

Example 2–1 Getting the Data Type of an MdmSource

```
public String getDataType(DataProvider dp, MdmSource metaSource)
{
   String theDataType = null;
   FundamentalMetadataProvider fmp =
                              dp.getFundamentalMetadataProvider();

   if (fmp.getBooleanDataType() == metaSource.getDataType())
       theDataType = "Boolean";
    else if (fmp.getDateDataType() == metaSource.getDataType())
       theDataType = "Date";
    else if (fmp.getDoubleDataType() == metaSource.getDataType())
       theDataType = "Double";
    else if (fmp.getFloatDataType() == metaSource.getDataType())
       theDataType = "Float";
```

```
else if (fmp.getIntegerDataType() == metaSource.getDataType())
   theDataType = "Integer";
else if (fmp.getShortDataType() == metaSource.getDataType())
   theDataType = "Short";
else if (fmp.getStringDataType() == metaSource.getDataType())
   theDataType = "String";
else if (fmp.getNumberDataType() == metaSource.getDataType())
   theDataType = "Number";
else if (fmp.getValueDataType() == metaSource.getDataType())
   theDataType = "Value";

return theDataType;
}
```

Type of MDM Metadata Objects

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

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

```
mdmCustDim
      mdmShipmentsRollup
            mdmTotalCust
            mdmRegion
            mdmWarehouse
            mdmShipTo
```

Because of the hierarchical structure, `mdmWarehouse` (for example) draws its elements from the elements of `mdmShipmentsRollup`. That is, the set of elements for `mdmWarehouse` corresponds to a subset of elements from `mdmShipmentsRollup`, and `mdmShipmentsRollup` is the type of `mdmWarehouse`.

Similarly, mdmShipmentsRollup is a component hierarchy of mdmCustDim. Therefore, mdmShipmentsRollup draws its elements from mdmCustDim, which is its type.

However, mdmCustDim is not a component of any other object. It represents the entire dimension. The pool of elements from which mdmCustDim draws its elements is the entire set of possible String values. Therefore, the type of mdmCustDim is the FundamentalMetadataObject that represents the OLAP 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 MdmLevel 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 its OLAP API data type. Typically, this is the String data type.
- The type of an MdmMeasure is the FundamentalMetadataObject that represents its OLAP API data type. Typically, this is one of the OLAP API numeric data types.

Getting the Type of an MdmSource

If you have obtained an MdmSource from the data store, and you want to find out its type, you call its getType method. This method returns the object that is the type of the MdmSource object.

For example, the following Java statement obtains the type of the MdmLevel named mdmWarehouse.

Example 2–2 Getting the Type of an MdmSource

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

3

Connecting to a Data Store

This chapter explains the procedure for connecting to a data store through the OLAP API.

This chapter includes the following topics:

Overview of the Connection Process

When an application gains access to data through the OLAP API, it uses a connection provided by the Oracle implementation of the Java Database Connectivity (JDBC) API from Sun Microsystems. For information about using this JDBC implementation, see the *Oracle Database JDBC Developer's Guide and Reference*.

The Oracle JDBC classes that you use to establish a connection to Oracle OLAP are in the `classes12.jar` file. For information about getting the JDBC Java archive (jar) file, see Appendix A, "Setting Up the Development Environment".

Connection Steps

The procedure for connecting involves loading an Oracle JDBC driver, getting a connection through that driver, and creating two OLAP API objects that handle transactions and data transfer.

These steps are described in the topic "Establishing a Connection" on page 3-2.

Prerequisites for Connecting

Before attempting to make an OLAP API connection to an Oracle database, 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 on which the data store is based.
- The Oracle JDBC and OLAP API jar files are on your application development computer and are accessible to the application code. 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 `TransactionProvider`.
4. Create a `DataProvider`.

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

Note that the `TransactionProvider` and `DataProvider` objects that you create in these steps are the ones that you use throughout your work with the data store. For example, when you create certain `Source` objects, you use methods of this `DataProvider` object.

Step 1: Load the JDBC Driver

The following line of 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 the *Oracle Database JDBC Developer's Guide and Reference*.

Step 2: Get 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 the *Oracle Database JDBC Developer's Guide and Reference* for details.

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

Step 3: Create a TransactionProvider

`TransactionProvider` is an OLAP API interface that is implemented for Oracle OLAP by the `ExpressTransactionProvider` concrete class. In your code, you create an instance of `ExpressTransactionProvider`, as in the following example.

Example 3–3 Creating a TransactionProvider

```
ExpressTransactionProvider tp = new ExpressTransactionProvider();
```

A `TransactionProvider` is required for creating a `DataProvider`.

Step 4: Create a DataProvider

`DataProvider` is an OLAP API abstract class. The concrete class `ExpressDataProvider` extends `DataProvider`. The following lines of code create and initialize an `ExpressDataProvider`.

Example 3–4 Creating a DataProvider

```
ExpressDataProvider dp = new ExpressDataProvider(conn, tp);
try
{
  dp.initialize();
}
catch(SQLException e)
{
  System.out.println("Could not initialize the DataProvider. " + e);
}
```

A `DataProvider` is required for creating a `MetadataProvider`, which is described in Chapter 4, "Discovering the Available Metadata".

Getting an Existing Connection

If you need access to the JDBC `OracleConnection` object after the connection has been established, you can call the `getConnection` method of your `ExpressDataProvider`. The following line of code calls the `getConnection` method of `dp`, which is an `ExpressDataProvider`.

Example 3–5 Getting an Existing Connection

```
oracle.jdbc.OracleConnection currentConn = dp.getConnection();
```

Executing DML Commands Through the Connection

Some applications depend on the run-time execution of Oracle OLAP data manipulation language (DML) commands or programs. DML commands and programs execute in an analytic workspace outside the context of MDM metadata, which is intrinsic to the OLAP API. Therefore, such commands and programs do not operate on MDM objects, such as an `MdmMeasure` or an `MdmDimension`. Instead, they operate on DML objects, such as a variable or a dimension. The MDM and DML contexts are related but distinct.

To execute DML commands or programs in an analytic workspace, create an OLAP API `SPLExecutor` object, specifying the JDBC `OracleConnection` object that you want to use. Note that the data manipulation language is sometimes referred to as a stored procedure language (SPL).

Example 3–6 creates and initializes an SPLExecutor object with a JDBC OracleConnection object called conn. With the `executeCommand` method of the `SPLExecutor` object, it passes to Oracle OLAP a DML command that attaches an analytic workspace named `myworkspace`.

For the complete code for the following example, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

Example 3–6 Executing DML Commands

```
SPLExecutor dmlExec = new SPLExecutor(conn);
try
{
  dmlExec.initialize();
}
catch(SQLException e)
{
  System.out.println("Cannot initialize the SPL executor. " + e);
}
String returnVal = dmlExec.executeCommand('aw attach myworkspace');
```

For information about using the DML, see the *Oracle OLAP DML Reference*. For more information about using an `SPLExecutor`, see the *Oracle OLAP Java API Reference*.

Closing a Connection

If you are finished using the OLAP 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 API resources. In the following code, the `DataProvider` is named `dp`.

```
dp.close();
```

When you have completed your work with the data store, use the `close` method of the JDBC `OracleConnection` object. In the following example, `conn` is the `OracleConnection` object.

Example 3–7 Closing a Connection

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

4

Discovering the Available Metadata

This chapter explains the procedure for discovering the metadata in a data store through the OLAP API.

This chapter includes the following topics:

For the complete code of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

Overview of the Procedure for Discovering Metadata

The OLAP API provides access to a collection of Oracle data for which a database administrator has created OLAP Catalog metadata. This collection of data is the data store for the application. The API also provides the ability to create custom metadata objects and map relational table data to the metadata objects, and to create queries that use the data to which the custom objects are mapped.

Potentially, the data store includes all of the measure folders that were created by the database administrator in the OLAP Catalog. 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

sees all of the measure folders (as `MdmSchema` objects) that the database administrator or the application created, but the user sees the measures and dimensions that are contained in those measure folders only if he or she has access rights to the relational tables to which the measures and dimensions are mapped.

MDM Metadata

When the database administrator created the OLAP Catalog metadata, the administrator created measures, dimensions, and other OLAP metadata objects by mapping them to columns in database tables or views. In the OLAP API, these objects are accessed as multidimensional metadata (MDM) objects, as described in Chapter 2, "Understanding OLAP API Metadata".

The mapping between the OLAP metadata objects and the MDM objects is automatically performed by Oracle OLAP. You can, however, discover the mapping of the MDM objects by using classes in the `oracle.olapi.metadata.mtm` package, as described in Chapter 5, "Working with Metadata Mapping Objects". You can also create your own custom MDM metadata objects and map them to database tables or views using MTM objects.

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 its characteristics are.

Therefore, after connecting, your first step is to find out what metadata is available. Armed with this knowledge, you can present choices to the end user about what data should be selected or calculated and how it should be displayed.

Steps in Discovering the Metadata

Before investigating the metadata, your application must make a connection to Oracle OLAP, as described in Chapter 3, "Connecting to a Data Store". Then, your application might perform the following steps:

1. Create an `MdmMetadataProvider`.
2. Get the root `MdmSchema` from the `MdmMetadataProvider`.
3. Get the contents of the root `MdmSchema`, which include `MdmMeasure`, `MdmDimension`, `MdmMeasureDimension`, and `MdmSchema` objects. In addition, get the contents of any subschemas.

4. Get the components or related objects of each `MdmMeasure` and `MdmDimension`. For example, get the `MdmDimension` objects for each `MdmMeasure`, and for each `MdmDimension` get its `MdmHierarchy` objects.

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

Discovering Metadata and Making Queries

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 MDM objects. These `Source` objects are referred to as primary `Source` objects. `Source` objects specify the data for querying.

This chapter focuses on the initial step of discovering the available metadata, but it also briefly mentions the step of getting a primary `Source` from a metadata object. Subsequent chapters of this guide explain how you work with primary `Source` objects and create queries based on them.

Creating an MdmMetadataProvider

An `MdmMetadataProvider` gives access to the metadata in a data store. It provides OLAP metadata objects, such as measures, dimensions, and measure folders, as corresponding MDM objects, such as `MdmMeasure`, `MdmDimension`, and `MdmSchema` objects.

Before you can create an `MdmMetadataProvider`, you must create a `DataProvider` as described in Chapter 3, "Connecting to a Data Store". Example 4–1 creates an `MdmMetadataProvider`. In the example, `dp` is an `ExpressDataProvider`.

Example 4–1 Creating an MdmMetadataProvider

```
MdmMetadataProvider mp = null;
try
{
  mp = (MdmMetadataProvider) dp.getDefaultMetadataProvider();
}
catch (UnsupportedDatabaseException e)
{
  System.out.println("Cannot create the MDM metadata provider. " + e);
}
```

Getting the Root MdmSchema

Getting the root `MdmSchema` is the first step in exploring the metadata in your data store.

Function of the Root MdmSchema

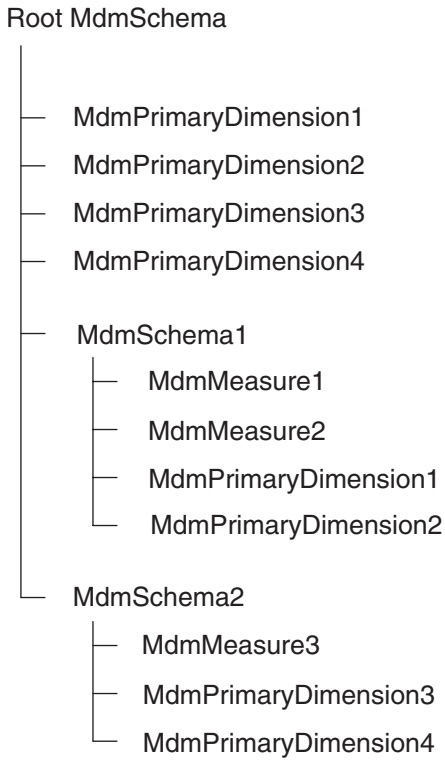
The metadata objects that are accessible through an `MdmMetadataProvider` are organized in a tree-like structure, with the root `MdmSchema` at the top. Under the root `MdmSchema` are `MdmPrimaryDimension` objects and one or more `MdmSchema` objects, which are referred to as subschemas. In addition, if an `MdmMeasure` object does not belong to any subschema, then it is included under the root.

Subschemas have their own `MdmMeasure` and `MdmPrimaryDimension` objects. Optionally, they can have their own subschemas as well.

The root `MdmSchema` contains all of the `MdmPrimaryDimension` objects that are in the subschemas. Therefore, an `MdmPrimaryDimension` typically appears twice in the tree. It appears once under the root `MdmSchema` and again under the subschema. If an `MdmPrimaryDimension` does not belong to a subschema, then it is listed only under the root.

The starting point for discovering the available metadata objects is the root `MdmSchema`, which is the top of the tree. The following diagram illustrates an `MdmSchema` that has two subschemas and four `MdmPrimaryDimension` objects.

Figure 4–1 Root MdmSchema and Subschemas



In the OLAP Catalog, a database administrator arranges dimensions and measures under one or more top-level measure folders. When Oracle OLAP maps the measure folders to `MdmSchema` objects, it always creates the root `MdmSchema` over the `MdmSchema` objects for the top-level measure folders. Therefore, even if the database administrator creates only one measure folder, its corresponding `MdmSchema` is a subschema under the root.

For more information about MDM metadata objects and how they map to OLAP metadata objects, see Chapter 2, "Understanding OLAP API Metadata".

Calling the getRootSchema Method

The following code gets the root MdmSchema for mp, which is an MdmMetadataProvider.

Example 4–2 Getting the Root MdmSchema

```
MdmSchema rootSchema = mp.getRootSchema();
```

Getting the Contents of the Root MdmSchema

The root MdmSchema contains MdmPrimaryDimension objects, MdmSchema objects, and possibly MdmMeasure objects. In addition, the root MdmSchema has an MdmMeasureDimension that has a List of all of the MdmMeasure objects.

Getting the MdmDimension Objects in an MdmSchema

The following code gets a List of the MdmPrimaryDimension objects that are in rootSchema, which is the root MdmSchema. The List does not include the MdmMeasureDimension.

Example 4–3 Getting MdmDimension Objects

```
List dims = rootSchema.getDimensions();
```

Getting the Subschemas in an MdmSchema

The following code gets a List of MdmSchema objects that are in rootSchema.

Example 4–4 Getting Subschemas

```
List subSchemas = rootSchema.getSubSchemas();
```

Getting the Contents of Subschemas

For each MdmSchema that is under the root MdmSchema, you can call its getMeasures, getDimensions, and getSubSchemas methods. The procedures are the same as those for getting the contents of the root MdmSchema.

Getting the MdmMeasureDimension and Its Contents

Example 4–5 gets the MdmMeasureDimension that is in the root MdmSchema. Use this method only on the root MdmSchema, because only the root MdmSchema has the MdmMeasureDimension. The example displays the names of the MdmMeasure objects that are contained by the MdmMeasureDimension.

Example 4–5 Getting the MdmMeasureDimension and Its Contents

```
MdmMeasureDimension mdmMeasureDim =
             (MdmMeasureDimension) rootSchema.getMeasureDimension();
List mdmMeasureDimMeasures = mdmMeasureDim.getMeasures();
Iterator mdmMeasureDimMeasuresItr = mdmMeasureDimMeasures.iterator();
MdmMeasure measure = null;
System.out.println("The measures in the MdmMeasureDimension are:");
while (mdmMeasureDimMeasuresItr.hasNext())
{
  measure = (MdmMeasure) mdmMeasureDimMeasuresItr.next();
  System.out.println("\t" + measure.getName());
}
```

Getting the Characteristics of Metadata Objects

Having discovered the list of MdmMeasure and MdmDimension objects, the next step in metadata discovery involves finding out the characteristics of those objects.

Getting the MdmDimension Objects for an MdmMeasure

A primary characteristic of an MdmMeasure is that it has related MdmPrimaryDimension objects. Example 4–6 gets a List of MdmPrimaryDimension objects for mdmUnits, which is an MdmMeasure.

Example 4–6 Getting the Dimensions of an MdmMeasure

```
List dimsOfUnits = mdmUnits.getDimensions();
```

The getMeasureInfo method, which is in the Example 4–9, shows one way to iterate through the MdmPrimaryDimension objects belonging to an MdmMeasure.

Getting the Related Objects for an MdmPrimaryDimension

An MdmPrimaryDimension has one or more component MdmHierarchy objects, which you can obtain by calling its getHierarchies method. That method

returns a `List` of `MdmHierarchy` objects. If an `MdmHierarchy` is an `MdmLevelHierarchy`, then it has levels that you can obtain by calling its `getLevels` method.

Example 4–7 demonstrates how you can get the `MdmHierarchy` objects for an `MdmPrimaryDimension`. The example displays the OLAP Catalog names of the `MdmHierarchy` objects.

Example 4–7 Getting the MdmHierarchy Components of an MdmPrimaryDimenison

```
List mdmHiers = mdmPrimaryDim.getHierarchies();
Iterator mdmHiersItr = mdmHiers.iterator();
System.out.println("The MdmHierarchy components of " + mdmPrimaryDim.getName()
                   + " are:");
while (mdmHiersItr.hasNext())
{
  MdmHierarchy mdmHier = (MdmHierarchy) mdmHiersItr.next();
  System.out.println("\t" + mdmHier.getName());
}
```

The `getDimInfo` method in Example 4–9 shows one way to get the following metadata objects for an `MdmDimension`.

- Its concrete class.
- Its `MdmHierarchy` objects.
- Its default `MdmHierarchy` object.
- The `MdmAttribute` objects returned by its `getAttributes` method.
- Its parent, ancestors, level, and level depth attributes.

Methods are also available for obtaining other `MdmPrimaryDimension` characteristics. See the *Oracle OLAP Java API Reference* for descriptions of all of the methods of the MDM classes.

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. Its function is informational, recording the existence, structure, and characteristics of the data. It does not give access to the data values.

In order 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 called 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 4–8 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 6, "Understanding Source Objects".

Sample Code for Discovering Metadata

The sample code that follows is a simple Java program called SampleMetadataDiscoverer10g. The program discovers the metadata objects that are under the root MdmSchema of a data store. The output of the program lists the names and related objects for the MdmMeasure and MdmDimension objects in the root MdmSchema and the MdmSchema subschema for the Global relational schema.

After presenting the program code, this topic presents the output of the program when it is run against a data store that consists of the Global relational schema. In the OLAP metadata, the Global schema is represented as the GLOBAL_CAT measure folder. Through an OLAP API connection, the GLOBAL_CAT measure folder maps to an MdmSchema whose name is GLOBAL_CAT.

Code for the SampleMetadataDiscoverer10g Program

The program in Example 4–9 gets the OLAP Catalog metadata objects that map to the tables and views of the Global relational schema. It passes the command line arguments that specify the server on which the Oracle Database instance is running and a user name and password to the connectToDB method of a MyConnection10g object, which establishes a connection to the database.

The code for the MyConnection10g is not shown here, but the procedure for connecting is described in Chapter 3, "Connecting to a Data Store". The complete code for the MyConnection10g and the SampleMetadataDiscoverer10g classes is available from the Overview of the *Oracle OLAP Java API Reference*.

Example 4–9 Discovering the OLAP Catalog Metadata

```
package globalExamples;

import oracle.express.olapi.data.full.ExpressDataProvider;
import oracle.olapi.metadata.mdm.*;
import oracle.olapi.data.source.Source;

import java.util.List;
import java.util.Iterator;

/**
 * Discovers the MDM metadata objects in the Global schema.
 * This file and the MyConnection10g.java file are available from the
 * Overview of the <I>Oracle OLAP Java API Reference</I>.
 */
public class SampleMetadataDiscoverer10g
{
/**
 * Constant to use to display less information about metadata objects.
 */
static final int TERSE = 0;
/**
 * Constant to use to display more information about metadata objects.
 */
static final int VERBOSE = 1;

private MdmSchema root = null;
private MdmPrimaryDimension mdmDim = null;

  public SampleMetadataDiscoverer10g()
  {
  }

  /**
   * Creates an object that makes a connection to an Oracle database
   * and gets MDM metadata objects.
   */
  public void run(String[] args)
  {
    // Connect through JDBC to an instance of an Oracle database
    // and get a DataProvider.
    MyConnection10g myConn = new MyConnection10g();
    ExpressDataProvider dp = myConn.connectToDB(args, TERSE);

    // Get the default MdmMetadataProvider from the DataProvider.
```

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

  // Get metadata information about the root MdmSchema and its subschemas.
  try
  {
    root = mp.getRootSchema();
    System.out.println("The root MdmSchema is " + root.getName() + ".\n");
    getSchemaInfo(root, VERBOSE);
  }
  catch (Exception e)
  {
    System.out.println("Encountered exception. " + e);
  }

  // Get the Source for the dimension that was saved in getDimInfo.
  System.out.println("\nMaking a Source object for dimension " +
                     mdmDim.getName() + ".");
  Source dimSource = mdmDim.getSource();
  System.out.println("Made the Source.");

  // Close the ExpressDataProvider and the connection.
  dp.close();
  System.out.println("\nClosed the DataProvider.");
  myConn.closeConnection();
  System.out.println("Closed the connection.");
}

/**
 * Gets information about an MdmSchema.
 */
public void getSchemaInfo(MdmSchema schema, int outputStyle)
{
  if (schema == root)
  {
    System.out.println("The MdmPrimaryDimension components of" +
                       "the root schema are:");
  }
```

```
else
{
  System.out.println("   The MdmPrimaryDimension components of schema "
                     +  schema.getName() + " are:");
}
// Get the dimension information for the MdmSchema.
MdmPrimaryDimension oneDim = null;
int i = 1;
try
{
  List dims = schema.getDimensions();
  Iterator dimIter = dims.iterator();
  // Save the first dimension to use later for getting its Source.
  mdmDim = (MdmPrimaryDimension) dims.get(0);
  // Iterate through the list of MdmPrimaryDimension objects and get
  // information about each one.
  while (dimIter.hasNext())
  {
    oneDim = (MdmPrimaryDimension) dimIter.next();
    getDimInfo(i, oneDim, outputStyle);
    i++;
  }
}
catch (Exception e)
{
  System.out.println("   Encountered exception. " + e);
}

// If the MdmSchema is the root MdmSchema, get the
// MdmMeasureDimension amd get its measures.
MdmMeasure oneMeasure = null;
MdmMeasureDimension mdmMeasureDim =
                      (MdmMeasureDimension) schema.getMeasureDimension();
if (mdmMeasureDim != null)
{
  System.out.println("The measures of the MdmMeasureDimension are:");
  List mdmMeasures = mdmMeasureDim.getMeasures();
  Iterator mdmMeasuresIter = mdmMeasures.iterator();
  while (mdmMeasuresIter.hasNext())
  {
    oneMeasure = (MdmMeasure) mdmMeasuresIter.next();
    getMeasureInfo(oneMeasure, outputStyle);
    System.out.println(" ");
  }
}
```

```
  // Get the measures from the MdmSchema.
  try
  {
    List mdmMeasures = schema.getMeasures();
    if (mdmMeasures.size() > 0)
    {
      Iterator mdmMeasuresIter = mdmMeasures.iterator();
       System.out.println("\n   The measures of schema " +
                          schema.getName() + " are:");
      while (mdmMeasuresIter.hasNext())
      {
        oneMeasure = (MdmMeasure) mdmMeasuresIter.next();
        getMeasureInfo(oneMeasure, outputStyle);
      }
    }
  }
  catch (Exception e)
  {
      System.out.println("   Encountered exception. " + e);
  }

  // Get the subschema information for the MdmSchema.
  MdmSchema oneSchema = null;
  try
  {
    List subSchemas = schema.getSubSchemas();
    Iterator subSchemaIter = subSchemas.iterator();
    while (subSchemaIter.hasNext())
    {
      oneSchema = (MdmSchema) subSchemaIter.next();
      // To get information on subschemas other than the Global
      // schema, GLOBAL_CAT, remove the if condition and call
      // the getSchemaInfo method.
      if (oneSchema.getName().equals("GLOBAL_CAT"))
        getSchemaInfo(oneSchema, TERSE);
    }
  }
  catch (Exception e)
  {
      System.out.println("   Encountered exception. " + e);
  }
}
```

```
/**
 * Gets information about an MdmMeasure.
 */
public void getMeasureInfo(MdmMeasure measure, int outputStyle)
{
  System.out.println("   " + measure.getName());
  if (outputStyle == VERBOSE)
  {
    // Get the dimensions of the MdmMeasure.
    try
    {
      List mDims = measure.getDimensions();
      Iterator mDimIter = mDims.iterator();
      System.out.println("   Its dimensions are: ");
      while (mDimIter.hasNext())
      {
          System.out.println("   " +
                      ((MdmDimension) mDimIter.next()).getName());
      }
    }
    catch (Exception e)
    {
      System.out.println("   Encountered exception. " + e);
    }
  }
}

/**
 * Gets information about an MdmDimension.
 */
public void getDimInfo(int count,
                       MdmPrimaryDimension dim,
                       int outputStyle)
{
  if (outputStyle == TERSE)
    System.out.println("   " + dim.getName());

  else if (outputStyle == VERBOSE)
  {
    System.out.println(count + ": MdmPrimaryDimension Name: " +
                       dim.getName());
    String description = dim.getDescription();
    if (description.length() > 0)
      System.out.println("   Description: " + dim.getDescription());
```

```
    // Determine the type of the MdmPrimaryDimension.
    try
    {
      if (dim instanceof MdmStandardDimension)
      {
        System.out.println("   It is an MdmStandardDimension.");
      }
      else if (dim instanceof MdmTimeDimension)
      {
        System.out.println("   It is an an MdmTimeDimension.");
      }
      else if (dim instanceof MdmMeasureDimension)
      {
        System.out.println("   It is an MdmMeasureDimension.");
      }
    }
    catch (Exception e)
    {
      System.out.println("   Encountered exception. " + e);
    }

    // Get the attributes of the MdmPrimaryDimension
    System.out.println("   Its attributes are:");
    try
    {
      List attributes = dim.getAttributes();
      Iterator attrIter = attributes.iterator();
      while (attrIter.hasNext())
      {
        System.out.println("   Attribute: " +
                           ((MdmAttribute) attrIter.next()).getName());
      }
    }
    catch (Exception e)
    {
      System.out.println("   Encountered exception. " + e);
    }

    // Get the hierarchies of the MdmPrimaryDimension
    getHierInfo(dim);
    System.out.println(" ");
  }
}
```

```
/**
 * Gets the MdmHierarchy components of an MdmPrimaryDimension.
 */
public void getHierInfo(MdmPrimaryDimension dim)
{
  List mdmHiers = dim.getHierarchies();
  Iterator mdmHiersItr = mdmHiers.iterator();
  MdmHierarchy mdmHier = null;
  MdmLevelHierarchy mdmLevelHier = null;
  boolean isLevelHierarchy = false;
  int i = 1;
  System.out.println("   The MdmHierarchy components of " +
                     dim.getName() + " are:");
  while (mdmHiersItr.hasNext())
  {
    mdmHier = (MdmHierarchy) mdmHiersItr.next();
    System.out.println("   "+ i + ": " + mdmHier.getName());
    if (mdmHier.isDefaultHierarchy())
    {
      System.out.println("   "+ i + ": " + mdmHier.getName());
         " is the default MdmHierarchy of " + dim.getName()  + ".");
    }
    if (mdmHier instanceof MdmLevelHierarchy)
    {
      mdmLevelHier = (MdmLevelHierarchy) mdmHier;
      System.out.println("      It is an MdmLevelHierarchy.");
      isLevelHierarchy = true;
    }
    else if (mdmHier instanceof MdmValueHierarchy)
    {
      System.out.println("      It is an MdmValueHierarchy");
    }
    System.out.println("      Its attributes are:");
    if (isLevelHierarchy)
    {
      System.out.println("      Level attribute: "
                       + mdmLevelHier.getLevelAttribute().getName());
      System.out.println("      Level depth attribute: "
                        + mdmLevelHier.getLevelDepthAttribute().getName());
    }
    System.out.println("      Parent attribute: " +
                       mdmHier.getParentAttribute().getName());
    System.out.println("      Ancestors attribute: " +
                       mdmHier.getAncestorsAttribute().getName());
```

```
      if (isLevelHierarchy)
        getLevelInfo(mdmLevelHier);
      i++;
    }
  }

  /**
   * Gets the MdmLevel components of an MdmLevelHierarchy.
   */
  public void getLevelInfo(MdmLevelHierarchy mdmHier)
  {
    List mdmLevels = mdmHier.getLevels();
    Iterator mdmLevelsItr = mdmLevels.iterator();
    System.out.println("      Its levels are:");
    while (mdmLevelsItr.hasNext()) {
      MdmLevel mdmLevel = (MdmLevel) mdmLevelsItr.next();
      System.out.println("      " + mdmLevel.getName());
    }
  }

  /**
   * Creates a new SampleMetadataDiscoverer10g object and calls its
   * run method.
   */
  public static void main(String[] args)
  {
    new SampleMetadataDiscoverer().run(args);
  }
}
```

Output from the SampleMetadataDiscoverer10g Program

The output from the sample program consists of text lines produced by Java statements such as the following one.

```
System.out.println("The root MdmSchema is " + root.getName() + ".\n");
```

When the program is run on the Global schema, the output includes the following items:

- The name of the root `MdmSchema`, which is `ROOT`.
- The names and other information for the `MdmPrimaryDimension` objects in the root `MdmSchema`.

- The measures in the `MdmMeasureDimension`.
- The dimensions and measures of the `GLOBAL_CAT` `MdmSchema`.

  Because the `GLOBAL_CAT` `MdmSchema` is the only subschema under the root `MdmSchema`, its `MdmPrimaryDimension` objects are identical to those in the root.
- Two lines that indicate that the code got the primary `Source` for an `MdmPrimaryDimension`.

Here is the output. In order to conserve space, some blank lines have been omitted.

```
The root MdmSchema is ROOT.

The MdmPrimaryDimension components of the root schema are:
1: MdmPrimaryDimension Name: CHANNEL
   It is an MdmStandardDimension.
   Its attributes are:
   Attribute: Long_Description
   Attribute: Short_Description
   The MdmHierarchy components of CHANNEL are:
   1: CHANNEL_ROLLUP
      CHANNEL_ROLLUP is the default MdmHierarchy of CHANNEL.
      It is an MdmLevelHierarchy.
      Its attributes are:
      Level attribute: D_GLOBAL.CHANNELLEVEL_ATTRIBUTE
      Level depth attribute: D_GLOBAL.CHANNELLEVELDEPTH_ATTRIBUTE
      Parent attribute: D_GLOBAL.CHANNELPARENT_ATTRIBUTE
      Ancestors attribute: D_GLOBAL.CHANNELANCESTORS_ATTRIBUTE
      Its levels are:
      ALL_CHANNELS
      CHANNEL

2: MdmPrimaryDimension Name: CUSTOMER
   It is an MdmStandardDimension.
   Its attributes are:
   Attribute: Long_Description
   Attribute: Short_Description
   The MdmHierarchy components of CUSTOMER are:
   1: MARKET_ROLLUP
      It is an MdmLevelHierarchy.
      Its attributes are:
      Level attribute: D_GLOBAL.CUSTOMERLEVEL_ATTRIBUTE
      Level depth attribute: D_GLOBAL.CUSTOMERLEVELDEPTH_ATTRIBUTE
      Parent attribute: D_GLOBAL.CUSTOMERPARENT_ATTRIBUTE
      Ancestors attribute: D_GLOBAL.CUSTOMERANCESTORS_ATTRIBUTE
```

```
      Its levels are:
      TOTAL_MARKET
      MARKET_SEGMENT
      ACCOUNT
      SHIP_TO
   2: SHIPMENTS_ROLLUP
      SHIPMENTS_ROLLUP is the default MdmHierarchy of CUSTOMER.
      It is an MdmLevelHierarchy.
      Its attributes are:
      Level attribute: D_GLOBAL.CUSTOMERLEVEL_ATTRIBUTE
      Level depth attribute: D_GLOBAL.CUSTOMERLEVELDEPTH_ATTRIBUTE
      Parent attribute: D_GLOBAL.CUSTOMERPARENT_ATTRIBUTE
      Ancestors attribute: D_GLOBAL.CUSTOMERANCESTORS_ATTRIBUTE
      Its levels are:
      ALL_CUSTOMERS
      REGION
      WAREHOUSE
      SHIP_TO

3: MdmPrimaryDimension Name: PRODUCT
   It is an MdmStandardDimension.
   Its attributes are:
   Attribute: Long_Description
   Attribute: Package
   Attribute: Short_Description
   The MdmHierarchy components of PRODUCT are:
   1: PRODUCT_ROLLUP
      PRODUCT_ROLLUP is the default MdmHierarchy of PRODUCT.
      It is an MdmLevelHierarchy.
      Its attributes are:
      Level attribute: D_GLOBAL.PRODUCTLEVEL_ATTRIBUTE
      Level depth attribute: D_GLOBAL.PRODUCTLEVELDEPTH_ATTRIBUTE
      Parent attribute: D_GLOBAL.PRODUCTPARENT_ATTRIBUTE
      Ancestors attribute: D_GLOBAL.PRODUCTANCESTORS_ATTRIBUTE
      Its levels are:
      TOTAL_PRODUCT
      CLASS
      FAMILY
      ITEM

4: MdmPrimaryDimension Name: TIME
   It is an an MdmTimeDimension.
   Its attributes are:
   Attribute: End_Date
   Attribute: Long_Description
```

```
   Attribute: Short_Description
   Attribute: Time_Span
   The MdmHierarchy components of TIME are:
   1: CALENDAR
      CALENDAR is the default MdmHierarchy of TIME.
      It is an MdmLevelHierarchy.
      Its attributes are:
      Level attribute: D_GLOBAL.TIMELEVEL_ATTRIBUTE
      Level depth attribute: D_GLOBAL.TIMELEVELDEPTH_ATTRIBUTE
      Parent attribute: D_GLOBAL.TIMEPARENT_ATTRIBUTE
      Ancestors attribute: D_GLOBAL.TIMEANCESTORS_ATTRIBUTE
      Its levels are:
      YEAR
      QUARTER
      MONTH

The measures of the MdmMeasureDimension are:
   UNIT_COST
   Its dimensions are:
   PRODUCT
   TIME

   UNIT_PRICE
   Its dimensions are:
   PRODUCT
   TIME

   UNITS
   Its dimensions are:
   CHANNEL
   CUSTOMER
   PRODUCT
   TIME

Schema: GLOBAL_CAT
   The MdmPrimaryDimension components of schema GLOBAL_CAT are:
   CHANNEL
   CUSTOMER
   PRODUCT
   TIME

   The measures of schema GLOBAL_CAT are:
   UNITS
   UNIT_COST
   UNIT_PRICE
```

```
Making a Source object for dimension CHANNEL.
Made the Source

Closed the DataProvider.
Closed the connection.
```

5

Working with Metadata Mapping Objects

The objects in the MDM model, which is described in Chapter 2, are based on relational tables and views in the data store. Metadata mapping (MTM) objects provide the information that maps the MDM objects to the relational tables and views on which the MDM objects are based. MTM objects are instances of the classes in the `oracle.olapi.metadata.mtm` package.

Application developers who have extensive experience with the OLAP API and with SQL can investigate, and in some cases create, objects from the MTM classes. For example, they might want to investigate MTM objects in order to discover the tables and columns to which particular MDM objects are mapped. Or they might want to create new objects in order to implement custom MDM objects, such as an `MdmMeasure`.

This chapter briefly describes the MTM objects, explains key concepts required for understanding them, and provides simple examples of how they can be used. The chapter has the following sections:

- Overview of the MTM Classes
- Discovering the Columns Mapped To an MdmSource
- Creating a Custom Measure
- Understanding Solved and Unsolved Data

For detailed information about the MTM classes, see the *Oracle OLAP Java API Reference*. For the complete code for the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

Overview of the MTM Classes

When an application developer uses Source objects to specify a query and Cursor objects to execute it, Oracle OLAP first identifies the MdmSource objects that correspond to the Source objects, and then identifies the MtmSourceMap objects that correspond to those MDMSource objects. An MtmSourceMap object maps the relationship between an MdmSource and the underlying SQL tables and expressions on which the MdmSource is based.

Oracle OLAP must identify the underlying SQL tables and expressions, because it must generate a SQL SELECT statement for every MdmSource that is referenced in an OLAP API query. The SELECT statements are constructed by the SQL generator component of Oracle OLAP.

SELECT Statements for MdmSource Objects

The SQL generator tailors a SQL statement to the subclass of MdmSource for which it is generating the SQL code.

- For an MdmSubDimension, the SQL statement is based on an MtmDimensionMap. The code includes the following three parts, one on each line.

  ```
  SELECT select-list-expression
  FROM source-table
  ORDER BY expression
  ```

- For an MdmMeasure, the SQL statement is based on an MtmMeasureMap. The code includes the following two parts, one on each line.

  ```
  SELECT select-list-expression
  FROM source-table
  ```

- An MdmAttribute does not have its own SQL statement. An MdmAttribute is associated with an MdmDimension, and it is based on the table or tables to which the MdmDimension is mapped. The columns for the MdmAttribute are part of the *select-list-expression* for the SELECT statement on which the MdmDimension is based. An MtmAttributeMap stores information about those columns.

An MdmDimensionMap, MdmMeasureMap, or MdmAttributeMap references the following MTM objects, which hold information about the parts of the generated SQL statement:

- An MtmExpression, which identifies an expression that the SQL generator can use as the *select-list-expression*.
- An MtmTabularSource, which identifies a logical table that the SQL generator can use as the *source-table* in the FROM clause. A *source-table* can be a table or view, a SELECT statement, or the join of a pair of tables.
- An MtmDimensionOrderSpecification, which holds information that the SQL generator can use to construct the *expression* for the ORDER BY clause of the SELECT statement for an MdmDimension.

For MdmMeasure objects, the SQL generator also uses the following MTM objects:

- MtmBaseCube objects, which record the dimensionality of the MdmMeasure objects for one set of the dimension hierarchies of an MtmPartionedCube.
- MtmCubeDimensionality objects, which store information about the fact table and dimension tables that must be joined to specify the data of an MdmMeasure. An MtmBaseCube has a set of MtmCubeDimensionality objects, one for each dimension of the measures of the cube.

Purpose of MTM Objects

Instances of the classes in the MTM package provide the information that the SQL generator needs to construct SELECT statements that implement OLAP API queries. The information is recorded in the form of MTM objects, such as MtmExpression, MtmTabularSource, and MtmCube.

As an application developer, you can interrogate MTM objects to discover the underlying relational tables and expressions. In some cases, you can use the information that you have discovered to create new MdmSource objects.

Measures, Cubes, and Hierarchies

For mapping purposes, every MdmMeasure belongs to a cube, and all of the MdmMeasure objects in a cube have the same dimensionality. That is, the values of the measures are specified by elements of the same set of MdmDimension objects. Thus, when you know the cube to which a measure belongs, you also know its dimensionality. From the point of view of mapping, the MTM model only has to record the dimensionality once for all the measures in a cube.

Dimensions can have multiple hierarchies, and the underlying data can be different for each hierarchy. Therefore, the MTM model emphasizes hierarchy mappings, which are more specific, rather than dimension mappings.

The MTM model also considers the fact that if a cube is made up of dimensions with multiple hierarchies, then the data can be different for each combination of hierarchies. Therefore, such a cube is partitioned into base cubes, each of which represents one hierarchy combination.

For example, the OLAP Catalog has a cube for the UNITS measure and its four dimensions, which are TIME, PRODUCTS, CHANNEL, and CUSTOMER. CUSTOMER has two hierarchies, MARKET_ROLLUP and SHIPMENTS_ROLLUP. The other three dimensions each have only one hierarchy, which are CALENDAR, PRODUCT_ROLLUP, and CHANNEL_ROLLUP. The following are the possible hierarchy combinations for the measures.

CALENDAR, PRODUCT_ROLLUP, CHANNEL_ROLLUP, MARKET_ROLLUP

CALENDAR, PRODUCT_ROLLUP, CHANNEL_ROLLUP, SHIPMENTS_ROLLUP

The MtmPartitionedCube for that cube therefore has two MtmBaseCube objects.

For all of these reasons, the mappings that are recorded by MTM objects are organized by cube and hierarchy, rather than by measure and dimension.

Discovering the Columns Mapped To an MdmSource

Ordinarily, neither an end-user nor an application developer needs to know the names of the relational columns to which an MdmSource is mapped. However, sometimes this information can be useful. For example, an application developer might want to ask a database administrator (DBA) to change a particular value in a dimension or might want to identify an existing column so that the developer can map a new custom metadata object to it. To discover the columns to which an MdmSource is mapped, you use MTM objects.

To identify the columns to which an MdmSource is mapped, you first get the MtmSourceMap for the MdmSource, and then from it you get the MtmTabularSource. From the MtmTabularSource, you get the MtmColumnExpression objects that represent the columns.

Of course, not all MdmSource objects have a specific column that can be mapped. An MdmSource that is mapped to a specific column has an MtmExpression that is implemented as an MtmColumnExpression. The MtmExpression cannot be an MtmCustomExpression or MtmLiteralExpression, because these objects are not based on a specific column. You should be familiar with the data and metadata

with which you are working, so that you are not attempting to find specific columns for MdmSource objects that are derived or otherwise specified.

Example of Getting the Columns Mapped To an MdmLevelHierarchy

In Example 5–1, mdmProdHier is an MdmLevelHierarchy that represents the default hierarchy of the PRODUCT dimension. The example gets the MtmLevelHierarchyMap for the MdmLevelHierarchy, gets the MtmRdbmsTableOrView that represents the relational table to which the dimension is mapped, and then gets the MtmColumnExpression objects that represent the columns of the table.

Example 5–1 Getting the Columns for an MtmLevelHierarchyMap

```
MtmLevelHierarchyMap mtmProdHierMap =
           (MtmLevelHierarchyMap) mdmProdHier.getLevelHierarchyMap();
MtmRdbmsTableOrView mtmRdbmsTableOrView =
                      (MtmRdbmsTableOrView) mtmProdHierMap.getTable();
System.out.println("The name of the table is "
                   + mtmRdbmsTableOrView.getName());
List mdmProdColumns = mtmRdbmsTableOrView.getColumns();
Iterator mdmProdColItr = mdmProdColumns.iterator();
System.out.println("Its columns are:");
while (mdmProdColItr.hasNext())
{
  MtmColumnExpression mtmColExp = (MtmColumnExpression) mdmProdColItr.next();
  System.out.println(mtmColExp.getColumnName());
}
```

The output of the example is the following:

```
The name of the table is GLOBAL.PRODUCT_DIM
Its columns are:
TOTAL_PRODUCT_ID
CLASS_ID
FAMILY_ID
ITEM_ID
TOTAL_PRODUCT_DSC
FAMILY_DSC
ITEM_DSC
CLASS_DSC
ITEM_PACKAGE_ID
```

Example of Getting the Columns Mapped To an MdmLevel

In Example 5–2, the mdmShipToLevel object is the MdmLevel that represents the SHIP_TO level of the default MdmLevelHierarchy of the CUSTOMER dimension. The example gets the MtmColumnExpression object that represents the column to which the MdmLevel is mapped and then gets the table that the column is in.

Example 5–2 Getting the Column Mapped To an MdmLevel

```
MtmLevelMap mtmShipToLevelMap = mdmShipToLevel.getLevelMap();
MtmColumnExpression mtmShipToColumnExp = (MtmColumnExpression)
                                mtmShipToLevelMap.getLevelExpression();
String shipToLevelColumnName = mtmShipToColumnExp.getColumnName();
System.out.println(shipToLevelColumnName);

MtmRdbmsTableOrView mtmTableWithShipTo = (MtmRdbmsTableOrView)
                                      mtmShipToColumnExp.getTable();
System.out.println(mtmTableWithShipTo.getName());
```

The example displays the following:

```
SHIP_TO_ID
GLOBAL.CUSTOMER_DIM
```

Example of Getting the Columns Mapped To an MdmMeasure

In Example 5–3, mdmUnits is an MdmMeasure that represents the UNITS measure. The example gets the MtmMeasureMap for the MdmMeasure, gets the MtmPartitionedCube that represents the cube to which the measure belongs, and gets the base cubes of the MtmPartitionedCube. The base cubes are all instances of MtmUnsolvedCube.

For the first base cube, the example gets the MtmRdbmsTableOrView that represents the relational table to which the dimension is mapped, which is the GLOBAL.UNITS_HISTORY_FACT table, and then gets the MtmColumnExpression objects that represent the columns of the table.

The other base cube of the partitioned cube represents the other combination of dimension hierarchies for the cube. All of the base cubes are mapped to the same table.

Example 5–3 Getting the Columns For an MdmMeasure

```
MtmMeasureMap mtmMeasureMap = mdmUnits.getMeasureMap();
MtmPartitionedCube mtmPCube = (MtmPartitionedCube) mtmMeasureMap.getCube();
```

```
List baseCubes = mtmPCube.getBaseCubes();
MtmUnsolvedCube mtmFirstBaseCube = (MtmUnsolvedCube) baseCubes.get(0);
MtmRdbmsTableOrView mtmRdbmsTableorView =
                        (MtmRdbmsTableOrView) mtmFirstBaseCube.getTable();
System.out.println("The name of the table is " +
                    mtmRdbmsTableorView.getName());
List columns = mtmRdbmsTableorView.getColumns();
Iterator colItr = columns.iterator();
System.out.println("Its columns are:"
while (colItr.hasNext())
{
  MtmColumnExpression mtmColExpr = (MtmColumnExpression) colItr.next();
  System.out.println(mtmColExpr.getColumnName());
}
```

The example displays the following:

```
The name of the table is GLOBAL.UNITS_HISTORY_FACT
Its columns are:
CHANNEL_ID
SHIP_TO_ID
ITEM_ID
MONTH_ID
UNITS
```

Creating a Custom Measure

Using the MTM mapping objects, you can create a custom metadata objects, such as an `MdmMeasure`, that exists only for the life of your `MdmMetadataProvider`. A custom `MdmMeasure` must be assigned to an existing `MtmCube`.

To create a custom measure, you start with an existing `MdmMeasure` that has the dimensionality that you want your custom `MdmMeasure` to have. Oracle OLAP assigns the new `MdmMeasure` to the `MtmCube` to which the existing `MdmMeasure` belongs, and creates it within the scope of your current `MdmMetadataProvider`.

Complete the following steps to create the custom measure:

1. Call the `getMeasureMap` method of the existing `MdmMeasure`, which returns the `MtmMeasureMap` for the `MdmMeasure`.
2. Call the `getCube` method of the `MtmMeasureMap`, which returns the `MtmPartitionedCube` for `MtmMeasureMap`.
3. Call the `getMdmCustomObjectFactory` method of your `MdmMetadataProvider`, which returns an `MdmCustomObjectFactory`.

4. Call a method of the `MdmCustomObjectFactory` that creates a new `MtmExpression`.
5. Call a method of the `MdmCustomObjectFactory` that accepts the `MtmCube` and `MtmExpression` as parameters and returns a new custom `MdmMeasure`.

Example 5–4 demonstrates these steps. It creates a custom `MdmMeasure` that is based on the RDBMS column to which an existing `MdmMeasure` is mapped. In the example, the existing `MdmMeasure` is `mdmUnitPrice` and `mp` is the `MdmMetadataProvider`. The `MdmMeasure` is based on the `UNIT_PRICE` column of the `PRICE_AND_COST_HISTORY_FACT` table of the relational Global schema.

Example 5–4 Creating a Custom Measure

```
MtmMeasureMap mtmUnitPriceMap = mdmUnitPrice.getMeasureMap();
MtmPartitionedCube mtmUnitPricePartCube = (MtmPartitionedCube)
                                          mtmUnitPriceMap.getCube();
MdmCustomObjectFactory mdmCustObjFactory = mp.getMdmCustomObjectFactory();
FundamentalMetadataProvider fdp = dp.getFundamentalMetadataProvider();
FundamentalMetadataObject numberFMO = fdp.getNumberDataType();
MtmCustomExpression mtmCustExp =
   mdmCustObjFactory.createCustomExpression("UNIT_PRICE - UNIT_COST",
                                            numberFMO);
MdmMeasure mdmCustMeasure =
        mdmCustObjFactory.createNumericMeasure("MARKUP",
                                               mtmUnitPricePartCube,
                                               mtmCustExp);
```

Understanding Solved and Unsolved Data

The way in which an `MdmSource` is mapped by MTM objects depends on the way its underlying data is specified (the data might be solved or unsolved) as well as the form in which the data is aggregated (grouping set, rollup, or embedded totals form). An understanding of these storage and aggregation concepts can be useful when you peruse the MTM classes. Classes such as `MtmSolvedETCubeDimensionality` and `MtmUnsolvedLevelHierarchyMap` encapsulate the storage and aggregation types.

Solved Versus Unsolved Cubes and Hierarchies

Typically, the data that is analyzed using the OLAP API is structured hierarchically. Detailed (leaf-level) data is at the lowest level of the hierarchy, and aggregate data is at higher levels of the hierarchy. A hierarchy is one of two types:

- A level hierarchy, in which each element belongs to a level and the parent-child relationships are organized by level.
- A value hierarchy, in which each element participates in parent-child relationships but there are no levels in the logical organization. (However, in the MTM model, a logical value hierarchy is stored in a level-based form.)

The detail data is ordinarily specified by a DBA in relational tables, materialized views, or an analytic workspace. However, the aggregate data might, or might not, be specified by the DBA. Aggregate data that is not specified by the DBA must be calculated by Oracle OLAP.

If all the data for a cube is specified by the DBA, then the cube is considered to be **solved**. If some or all of the aggregate data must be calculated by Oracle OLAP, then the cube is **unsolved**.

Note that the data for a solved cube is not necessarily stored in the database. It might be specified by the DBA as a materialized view, which is calculated when necessary. The distinction between solved and unsolved cubes rests on who specifies the data: the DBA, or Oracle OLAP.

It is not only cubes that can be either solved or unsolved. Hierarchies can be solved or unsolved as well. If all of the elements of a hierarchy, both aggregate and detailed, exist in a single table or view, then the hierarchy is solved. If some or all of the aggregate elements must be collected by Oracle OLAP from separate tables, then the hierarchy is unsolved.

Aggregation Forms for Cubes

There are three possible forms in which data for a cube can be aggregated. For a solved cube, the DBA specified the method of aggregation, so the SQL statement that is constructed by the SQL generator does not have to reflect the aggregation form. However, for an unsolved cube, the Oracle OLAP SQL generator constructs a SQL statement that is appropriate to the aggregation form.

The following forms are supported. Each is described in terms of a SQL statement, though it might be specified by a DBA for a solved cube or by the SQL generator for an unsolved cube.

- **Grouping set aggregation form.** The SQL statement uses the `GROUP BY GROUPING SETS` syntax to aggregate the data for each level combination. The select list includes all of the level expressions as well as a `GROUPING_ID` expression for each hierarchy.
- **Rollup aggregation form.** The SQL statement uses the `GROUP BY ROLLUP` syntax to aggregate the data for each level combination. The select list includes all of the level expressions as well as a `GROUPING_ID` expression for each hierarchy.
- **ET aggregation form.** The SQL statement uses the `GROUP BY ROLLUP` syntax to aggregate the data for each level combination. However, only the ET and `GROUPING_ID` expression for each hierarchy are placed in the `SELECT` list.

The SQL expressions for the three aggregation forms are described in more detail in the rest of this section. For information about `GROUPING_ID` expressions, ET expressions, and the three aggregation forms, see the *Oracle OLAP Application Developer's Guide*.

Aggregation for Unsolved Cubes

Using the aggregation form for a given cube, the SQL generator constructs an appropriate SQL statement. (Note that the `getAggregationForm` method of an `MtmUnsolvedCube` returns its aggregation form.)

In all cases, the SQL statement aggregates the higher-level values from the detailed level (leaf-level) data. The statement has the following structure.

```
SELECT SUM(measure1), SUM(measure2), ..., SUM(measureN), dimension-keys
FROM fact-table, dimension-tables
WHERE join-condition
GROUP BY group-by-clause
```

For example, assume a single hierarchy with three levels: `Y` as the top level, `Q` as the middle level, and `M` as the bottom level. The `GROUP BY` clause is one of the following, depending on the aggregation form:

GROUP BY clause for grouping set aggregation:

```
GROUP BY GROUPING SETS((Y), (Q), (M))
```

GROUP BY clause for rollup aggregation:

```
GROUP BY Y, ROLLUP(Q, M)
```

GROUP BY clause for ET aggregation:

```
GROUP BY Y, ROLLUP(Q, M)
```

Using the same example, the *dimension-keys* component of the select list is one of the following, depending on the aggregation form:

***dimension-keys* for grouping set aggregation:**

```
SELECT Y, Q, M, GROUPING_ID(Y, Q, M)
```

***dimension-keys* for rollup aggregation:**

```
SELECT Y, Q, M, GROUPING_ID(Y, Q, M)
```

***dimension-keys* for ET aggregation:**

```
SELECT
   (CASE GROUPING_ID(Y, Q, M)
    WHEN 3 THEN Y
    WHEN 1 THEN Q
    ELSE M
    END) et_value,
   GROUPING_ID(Y, Q, M)
```

In general, for grouping set or rollup form, the *dimension-keys* component is made up of one expression for each level and one `GROUPING ID` expression. For ET form, the *dimension-keys* component is made up of an ET expression and a `GROUPING ID` expression.

Note that, in all cases, the *join-condition* in the generated SQL statement is determined by the `MtmUnsolvedCubeDimensionality` object that is associated with the `MtmUnsolvedCube`.

Aggregation for Solved Cubes

All of the values, both detail and aggregate, for a solved cube are explicitly specified by a DBA. Therefore, the `SELECT` statement that is generated by the Oracle OLAP SQL generator is relatively simple, and it has the same structure for all aggregation forms. Using the same example, the `SELECT` statement would be the following.

```
SELECT Y, Q, M, dimension-keys
FROM source-table
```

The *dimension-keys* component has the same make up as it does for unsolved cubes, varying by the aggregation form that the DBA used.

The DBA specifies the aggregation form for a solved cube when he or she is setting up metadata using Oracle Enterprise Manager or the SQL procedures provided by Oracle for working with the OLAP Catalog.

The DBA specifies one of the following two storage types:

- **ET (Embedded Totals).** The fact table for the cube includes all of the aggregated values for the associated measures. Therefore, materialized views are not required. DBAs can create cubes with the ET storage type when they use the SQL procedures for working with the OLAP Catalog.
- **Lowest Level.** The fact table for the cube includes only the detailed (leaf-level) data. Aggregated values must be supplied using materialized views. When DBAs create cubes using the OLAP Management tool in Oracle Enterprise Manager, the cubes are created with lowest level storage type. Using a different tool, the DBA can specify one of the following forms of materialized view for aggregating data:
  - **Grouping Set form**, in which all the hierarchy combinations are in a single materialized view. This form is created when the DBA uses the `DBMS_ODM` package procedures.
  - **Rolled Up form**, in which there is a separate materialized view for each hierarchy combination. This form is created when the DBA uses the OLAP Summary Adviser in Oracle Enterprise Manager.

Thus, there are three aggregation forms, which correspond to the following three concrete subclasses of the `MtmSolvedCubeDimensionality` class.

- `MtmSolvedETCubeDimensionality`
- `MtmSolvedGroupingSetCubeDimensionality`
- `MtmSolvedRollupCubeDimensionality`

A term that recurs in the methods of the MTM classes is GID, which stands for Grouping ID. It refers to the GID column of the fact table for a cube. The GID column, which is derived from the level columns in the fact table, identifies the level associated with each value in a hierarchy. The values of a GID column are calculated by assigning a zero to each non-null value and a one to each null value in the level columns. The resulting binary number is the value of the GID. Hierarchy values that have the same GID are in the same level.

For more information about storage types, aggregation forms, and GID columns, see the *Oracle OLAP Application Developer's Guide.*

Solve Specifications for Unsolved Cubes

In addition to the aggregation form, an `MtmUnsolvedCube` has an `MtmSolveSpecification`, which records the SQL operation that Oracle performs when it aggregates the measure data specified by a dimension, and the order in which Oracle aggregates the dimensions of the measure. An `MtmAggregationSpecification`, which is a subclass of `MtmSolveSpecification`, has one or more `MtmAggregationStep` objects.

An `MtmAggregationStep` specifies the SQL function and other aspects of operations to perform when Oracle aggregates the values of the measures of an `MtmUnsolvedCube` for the dimension hierarchies that of the `MtmDimensionMap` objects of the `MtmAggregationStep`. The `MtmDimensionMap` objects of an `MtmUnsolvedCube` are always instances of `MtmUnsolvedLevelHierarchyMap`.

Each `MtmUnsolvedLevelHierarchyMap` of the `MtmUnsolvedCube` is associated with one and only one `MtmAggregationStep`. An `MtmAggregationStep` can specify the aggregation operations for one or more of the `MtmUnsolvedLevelHierarchyMap` objects.

The default aggregation function is `SUM`. For an `MtmSimpleAggregationStep`, you can specify other SQL group functions or your own function. You can create a simple aggregation step or other types of aggregation steps with methods of an `MtmObjectFactory`, which you get from your `MdmMetadataProvider`.

The other types of aggregation steps are `MtmFirstLastAggregationStep`, `MtmWeightedAverageStep`, and `MtmNoAggregationStep`. An `MtmFirstLastAggregationStep` represents an aggregation that uses the `SUM` function and uses the measure data specified by the first or last child element of the current parent element as the aggregate measure value for the parent element. An `MtmWeightedAverageStep` specifies the `AVG` function with a weighting factor applied to the aggregation. An `MtmNoAggregationStep` specifies that no aggregation occur for the dimension hierarchy or hierarchies.

6

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 7, "Making Queries Using Source Methods", provides examples of using `Source` objects. Using `Template` objects to make modifiable queries is discussed in Chapter 11, "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

For the complete code for most of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

Overview of Source Objects

After you have used the classes in the `oracle.olapi.metadata.mdm` package to get `MdmSource` objects that represent measures and dimensions in the OLAP Catalog, 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 to get dimension or measure values from the database. To retrieve data from the database, you use a `Source` to create a `Cursor`.

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 primary methods and many shortcut methods that use one or more of the primary methods. The most complex primary 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 primary 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 elements that 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, thus producing derived `Source` objects in the process.
- Derived `Source` objects, which you get by calling methods on 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 API data types.
- List or range Source objects that you get by calling the createConstantSource, createListSource or createRangeSource methods of a DataProvider. Typically, you use this kind of Source as the joined or comparison parameter to the 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 createParameterizedSource methods of a DataProvider. 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.

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.

Data Type of a Source

As described in Chapter 2, "Understanding OLAP API Metadata", the OLAP 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 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 6–1 demonstrates getting the fundamental `Source` for the OLAP 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 6–1 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)
  System.out.println("The Source objects for the data types are all the same.");
else
  System.out.println("The Source objects for the data types are not " +
                     "all the same.");
```

The example displays the following:

```
The Source objects for the data types are all 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 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 its data type. 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 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:

- Its **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 its base Source as its type. An exception is the derived Source returned by the distinct method, whose type is the type of its base Source rather than the base Source itself.
- A fundamental Source. Methods such as position and count return a Source that has the fundamental Source for the OLAP API Integer data type as its type. Methods that make comparisons, such as eq, le, and so on, return a Source that has the fundamental Source for the Boolean data type as its 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 of a Source by calling its getType method.

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 6–2 the myList object is a list Source. The example uses myList to select values from prodRollup, a Source for the default MdmLevelHierarchy of the MdmPrimaryDimension for the PRODUCT dimension. In the example, dp is the DataProvider. Because prodSel is a subtype of prodRollup, the condition in the if statement is true.

Example 6–2 Using the isSubtypeOf Method

```
Source myList = dp.createListSource(new String[] {
                                    "PRODUCT_ROLLUP::FAMILY::4",
                                    "PRODUCT_ROLLUP::FAMILY::5",
                                    "PRODUCT_ROLLUP::FAMILY::7",
                                    "PRODUCT_ROLLUP::FAMILY::8"});
Source prodSel = prodRollup.selectValues(myList);
if (prodSel.isSubtypeOf(prodRollup))
  System.out.println("prodSel is a subtype of prodRollup.");
else
  System.out.println("prodSel is not a subtype of prodRollup.");
```

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

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 prodRollup, 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 prodRollup, 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 of a Source by calling its getID method. For example, the following code gets the identification of the Source for the MdmPrimaryDimension for the PRODUCT dimension and displays the value.

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

The preceding code displays the following:

```
The Source ID of prodDim is Hidden..D_GLOBAL.PRODUCT
```

The text displayed by Example 6–9 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 its `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

An input of a `Source` is also a `Source`. An input indicates that the values of the `Source` with the input 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 `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 `MdmDimensionedObject` subclasses `MdmMeasure` and `MdmAttribute`. They have inputs because the values of the 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 its base Source as an input.

For example, in the following code, the base Source is prodRollup. Its value method produces prodRollupValues, which has prodRollup as an input.

```
Source prodRollupValues = prodRollup.value();
```

The input provides the means to select values from prodRollup, as demonstrated by Example 6–2. The selectValues method in Example 6–2 is a shortcut for the following join method.

```
Source prodSel = prodRollup.join(prodRollup.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 prodRollup.value() method. The resulting unnamed Source has prodRollup as an input. The input is matched by the base of the join method, which is also prodRollup. The result of the join operation, prodSel, has the values of prodRollup that match the values of prodRollup that are in the comparison Source, myList.

If the joined Source were prodRollup and not the Source produced by prodRollup.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 prodRollup instead of having only the values of prodRollup 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 its getInputs method, the regular inputs by calling its getRegularInputs method, and its extraction inputs by calling its 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 its 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 its 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 its 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 its outputs because each data value of the measure is identified by a unique set of the values of its 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 its 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 6–3 produces a selection of the elements of shipRollup, 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 6–3 Using the join Method To Produce a Source Without an Output

```
Source custValuesToSelect = dp.createListSource(new String[]
                                    {"SHIPMENTS_ROLLUP::REGION::9",
                                     "SHIPMENTS_ROLLUP::REGION::10"});
Source shipRollupValues = shipRollup.value();
Source custSel = shipRollup.join(shipRollupValues,
                                 custValuesToSelect,
                                 Source.COMPARISON_RULE_SELECT,
                                 false);
```

The `shipRollupValues Source` has an input of `shipRollup`. In the `join` method in the example, the base `Source`, `shipRollup`, matches the input of the joined `Source`, `shipRollupValues` because the base and the input are the same object. The `join` method selects the elements of the base `shipRollup` whose values match the values of the joined `shipRollup` that are specified by the comparison `Source`, `custValuesToSelect`. The method produces a `Source`, `custSel`, that has only the selected elements of `shipRollup`. 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_ROLLUP::REGION::9` and `SHIPMENTS_ROLLUP::REGION::10`.

You produce a `Source` that has an output by specifying `true` as the `visible` parameter to the `join` method. Example 6–4 joins the `Source` objects for the dimension selections from Example 6–2 and Example 6–3 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 6–4, `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 6–4 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 its type, `custSel`, are in the right column.

Output Values	Type Values
PRODUCT_ROLLUP::FAMILY::4	SHIPMENTS_ROLLUP::REGION::10
PRODUCT_ROLLUP::FAMILY::4	SHIPMENTS_ROLLUP::REGION::9
PRODUCT_ROLLUP::FAMILY::5	SHIPMENTS_ROLLUP::REGION::10
PRODUCT_ROLLUP::FAMILY::5	SHIPMENTS_ROLLUP::REGION::9
PRODUCT_ROLLUP::FAMILY::7	SHIPMENTS_ROLLUP::REGION::10
PRODUCT_ROLLUP::FAMILY::7	SHIPMENTS_ROLLUP::REGION::9
PRODUCT_ROLLUP::FAMILY::8	SHIPMENTS_ROLLUP::REGION::10
PRODUCT_ROLLUP::FAMILY::8	SHIPMENTS_ROLLUP::REGION::9

The `custSelByProdSel Source` has two type elements, and its output 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 its `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 its `getOutputs` method, which returns a `List` of `Source` objects.

Example 6–5 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 6–5 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 level hierarchies of the dimensions, which are subtypes of the Source objects for the dimensions. It produces selections of the level 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 6–5 Using the join Method To Match Source Objects To Inputs

```
Source unitCost = mdmUnitCost.getSource();
Source calendar = mdmCalendar.getSource();
Source prodRollup = mdmProdRollup.getSource();
Source timeSel = calendar.join(calendar.value(),
                               dp.createListSource(new String[]
                                                   {"CALENDAR::MONTH::47",
                                                    "CALENDAR::MONTH::59"}),
                               Source.COMPARISON_RULE_SELECT,
                               false);
Source prodSel = prodRollup.join(prodRollup.value(),
                               dp.createListSource(new String[]
                                        {"PRODUCT_ROLLUP::ITEM::13",
                                         "PRODUCT_ROLLUP::ITEM::14",
                                         "PRODUCT_ROLLUP::ITEM::15"}),
                               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 its type, unitCost, that are specified by its 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 its Source. The unit cost values are formatted as dollar values.

```
        Output 1                Output 2          Type
         Values                  Values           Values
------------------------  ------------------  --------
PRODUCT_ROLLUP::ITEM::13  CALENDAR::MONTH::47  2897.40
PRODUCT_ROLLUP::ITEM::13  CALENDAR::MONTH::59  2376.73
PRODUCT_ROLLUP::ITEM::14  CALENDAR::MONTH::47  3238.36
PRODUCT_ROLLUP::ITEM::14  CALENDAR::MONTH::59  3015.90
PRODUCT_ROLLUP::ITEM::15  CALENDAR::MONTH::47  2847.47
PRODUCT_ROLLUP::ITEM::15  CALENDAR::MONTH::59  2819.85
```

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 API applications, the API provides shortcuts for these and many other join operations. Example 6–6 uses shortcuts for the join operations in Example 6–5 to produce the same result.

Example 6–6 Using Shortcuts

```
Source unitCost = mdmUnitCost.getSource();
StringSource calendar = (StringSource) mdmCalendar.getSource();
StringSource prodRollup =(StringSource) mdmProdRollup.getSource();
Source timeSel = calendar.selectValues(new String[]
                                       {"CALENDAR::MONTH::47",
                                        "CALENDAR::MONTH::59"}),
Source prodSel = prodRollup.selectValues(new String[]
                                        {"PRODUCT_ROLLUP::ITEM::13",
                                         "PRODUCT_ROLLUP::ITEM::14",
                                         "PRODUCT_ROLLUP::ITEM::15"}),
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 6–7 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 6–7 Matching the Base Source to an Input of the Joined Source

```
Source myList = dp.createListSource(new String[]
                                    "PRODUCT_ROLLUP::FAMILY::4",
                                    "PRODUCT_ROLLUP::FAMILY::5",
                                    "PRODUCT_ROLLUP::FAMILY::7",
                                    "PRODUCT_ROLLUP::FAMILY::8"});
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 6–7, 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_ROLLUP::FAMILY::5
PRODUCT_ROLLUP::FAMILY::8
```

If the visible parameter in Example 6–7 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_ROLLUP::FAMILY::5
  4     PRODUCT_ROLLUP::FAMILY::8
```

Example 6–8 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 prodRollup, shipRollup, calendar, and chanRollup 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 prodRollup in the first line of Example 6–8 results in prodSel, which specifies selected product values. In that method, the joined Source is the result of the value method of prodRollup. The joined Source has the same elements as prodRollup, and it has prodRollup 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, prodRollup, matches the input of the joined Source. Because prodRollup is the input of the joined Source, the Source returned by the join method has only the elements of the base, prodRollup, 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 6–8 result in selections for the other three dimensions.

The join method of timeSel has custSel as the joined Source. Its 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 customers 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 6–8 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 prodRollup. The input for the customers 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 shipRollup. 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 6–8 Matching an Input of the Base Source to an Output of the Joined Source

```
Source prodSel = prodRollup.join(prodRollup.value(),
                                 dp.createListSource(new String[]
                                     {"PRODUCT_ROLLUP::FAMILY::4",
                                      "PRODUCT_ROLLUP::FAMILY::5"}),
                                 Source.COMPARISON_RULE_SELECT,
                                 false);
Source custSel = shipRollup.join(shipRollup.value(),
                                 dp.createListSource(new String[]
                                    {"SHIPMENTS_ROLLUP::REGION::9",
                                     "SHIPMENTS_ROLLUP::REGION::10"}),
                                 Source.COMPARISON_RULE_SELECT,
                                 false);
Source timeSel =  calendar.join( calendar.value(),
                                dp.createConstantSource(
                                              "CALENDAR::YEAR::4"),
                                Source.COMPARISON_RULE_SELECT,
                                false);
```

```
Source chanSel = chanRollup.join(chanRollup.value(),
                                 dp.createConstantSource(
                                      "CHANNEL_ROLLUP::CHANNEL::4"),
                                 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.

```
4
4
            Products
           ----------
Customers   4     5
---------  ---  ----
10         846  1748
9          215   439
```

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       Portable PCs  Desktop PCs
-------------   ------------  -----------
North America   846           1748
Europe          215            439
```

To demonstrate turning inputs into outputs, Example 6–9 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 its inputs. The inputs of the `Source` for the measure are the `Source` objects for the `MdmPrimaryDimension` objects that are the dimensions of the measure.

Example 6–9 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 its elements 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 6–9 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;
System.out.println("The inputs of " + units.getID() + " are:");
while(inputsItr.hasNext())
{
  input = (Source) inputsItr.next();
  System.out.println(i + ": " + input.getID());
  i++;
}

System.out.println(" ");
int setSize = inputs.size();
for(i = 0; i < (setSize + 1); i++)
{
  System.out.println(units.getID() + " has " + inputs.size() +
                  " inputs and " + outputs.size() + " outputs.");
  if (i < setSize)
  {
    input = defaultHiers.get(i);
    System.out.println("Joining " + input.getID() + " to "
                                  + units.getID());
    units = units.join(input);
    inputs = units.getInputs();
    outputs = units.getOutputs();
  }
}

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

The text displayed by the example is the following:

```
The inputs of Hidden..M_GLOBAL.UNITS_CUBE.UNITS are:
1: Hidden..D_GLOBAL.TIME
2: Hidden..D_GLOBAL.PRODUCT
3: Hidden..D_GLOBAL.CUSTOMER
4: Hidden..D_GLOBAL.CHANNEL

Hidden..M_GLOBAL.UNITS_CUBE.UNITS has 4 inputs and 0 outputs.
Joining Hidden..D_GLOBAL.PRODUCT.PRODUCT_ROLLUP to
        Hidden..M_GLOBAL.UNITS_CUBE.UNITS
Join.0 has 3 inputs and 1 outputs.
Joining Hidden..D_GLOBAL.CUSTOMER.SHIPMENTS_ROLLUP to Join.0
Join.1 has 2 inputs and 2 outputs.
Joining Hidden..D_GLOBAL.TIME.CALENDAR to Join.1
Join.2 has 1 inputs and 3 outputs.
Joining Hidden..D_GLOBAL.CHANNEL.CHANNEL_ROLLUP to Join.2
Join.3 has 0 inputs and 5 outputs.

The outputs of Join.3 are:
1: Hidden..D_GLOBAL.CHANNEL.CHANNEL_ROLLUP
2: Hidden..D_GLOBAL.TIME.CALENDAR
3: Hidden..D_GLOBAL.CUSTOMER.SHIPMENTS_ROLLUP
4: Hidden..D_GLOBAL.PRODUCT.PRODUCT_ROLLUP
```

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.3` is `Hidden..D_GLOBAL.CHANNEL.CHANNEL_ROLLUP`, and its last output is `Hidden..D_GLOBAL.PRODUCT.PRODUCT_ROLLUP`.

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 `createParameterizedSource` method of the `DataProvider` you are using. In creating the parameterized `Source`, you supply a `Parameter` object. The `Parameter` supplies the value that the parameterized `Source` specifies.

`Parameter` objects are similar to `CursorInput` objects in that you use them to specify an initial value for a `Source` that is part of a query. A typical use of both `Parameter` and `CursorInput` objects is to specify the page edges of a cube. Example 7–9 demonstrates using `Parameter` objects to specify page edges.

An advantage of `Parameter` objects over `CursorInput` objects is that with `Parameter` objects you can easily fetch from the server only the set of elements that you currently need. Example 7–16 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 its `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 6–10 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 its element values the `Source` objects for unit cost and unit price measures. The example creates a `StringParameter` object that has as its 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 `context` object has a method that displays the values of the `Cursor` with only the local value of the dimension elements.

Example 6–10 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.
StringSource measParamSrc = dp.createParameterizedSource(measParam);
```

```
// 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 prepare and commit the
// current transaction. These operations are not shown.

// Create a Cursor.
CursorManagerSpecification cMngrSpec =
                        dp.createCursorManagerSpecification(results);
SpecifiedCursorManager  spCMngr = dp.createCursorManager(cMngrSpec);
Cursor resultsCursor = spCMngr.createCursor();

// Display the results.
context.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.
context.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
---- ----------- ------- ---------------   ---------
 58    Apr-01      13    Envoy Standard    2360.78
 58    Apr-01      14    Envoy Executive   2952.85
 59    May-01      13    Envoy Standard    2376.73
 59    May-01      14    Envoy Executive   3015.90
```

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
58	Apr-01	13	Envoy Standard	2412.42
58	Apr-01	14	Envoy Executive	3107.65
59	May-01	13	Envoy Standard	2395.63
59	May-01	14	Envoy Executive	3147.85

7

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 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:

For the complete code of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

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 6, "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 7–1 lists the basic Source methods.

Table 7–1 The Basic Source Methods

Method	Description
alias	Produces a Source that has the same elements as its base Source, but has its base Source as its type.
distinct	Produces a Source that has the same elements as its base Source, except that any elements that are duplicated in the base appear only once in the derived Source.
extract	Produces a Source that has the same elements as its base Source, but that has its base Source as an extraction input.
join	Produces a Source that has the elements of its 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 its base Source, and that has its 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 its base Source, but that has its 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 its aliased dimension, and then compare the results.

Example 7–1 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::MONTH::55`, which identifies the month January, 2001, the `custSel` value is `SHIPMENTS_ROLLUP::SHIP_TO::52`, which identifies the Business Word San Jose customer, and the `prodSel` value is `PRODUCT_ROLLUP::ITEM::15`, 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 following join operations matches `chanAlias1` to the input of `unitsSel1` and matches `chanAlias1` to the input of `unitsSel2`.

Example 7–1 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 its 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 its 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 `All Channels` is not greater than the units sold for the row `All Channels` value but it is greater than the units sold for the `Direct Sales`, `Catalog`, and `Internet` row values.

```
                ---------- chanAlias1 ----------
chanAlias2    All Channels  Direct Sales  Catalog  Internet
------------  ------------  ------------  -------  --------
All Channels       false       false       false    false
Direct Sales       true        false       true     false
Catalog            true        false       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 7–2 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 7–2, `mktRollup` is a `StringSource` that represents the `MARKET_ROLLUP` hierarchy of the `CUSTOMER` dimension. The `mktRollupAncestors` object is the `Source` for the ancestors attribute of that hierarchy. To get a `Source` that represents the descendents of the ancestors, the example uses the `join` method to select, for each element of `mktRollupAncestors`, the elements of `mktRollup` that have the `mktRollupAncestors` element as their ancestor. The join operation matches the base `Source`, `mktRollup`, to the input of the ancestors attribute.

The resulting `Source`, `mktRollupDescendants`, however, still has `mktRollup` as an input because the `Source` produced by the `mktRollup.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 `mktRollup` selects the element of `mktRollup` that has the value `MARKET_ROLLUP::ACCOUNT::23`, which is the Business World account, and produces `selVal`. The `join` method of `mktRollupDescendants` uses `selVal` as the `comparison` parameter. The method produces `selValDescendants`, which is has the elements of `mktRollupDescendants` that are present in `mktRollup`, and that are also in `selVal`. The input of `mktRollupDescendants` is matched by the joined `Source` `mktRollup`. The `mktRollup` `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 7–2 Using the distinct Method

```
Source mktRollupDescendants =
                    mktRollup.join(mktRollupAncestors, mktRollup.value());
Source selVal = mktRollup.selectValue("MARKET_ROLLUP::ACCOUNT::23");
```

```
Source selValDescendants = mktRollupDescendants.join(mktRollup,
                                                     selVal,
                                                     false);
Source selValPlusDescendants = selVal.appendValues(selValDescendants)
                                     .distinct();
```

A `Cursor` for the `selValPlusDescendants` `Source` has the following values:

```
MARKET_ROLLUP::ACCOUNT::23
MARKET_ROLLUP::SHIP_TO::51
MARKET_ROLLUP::SHIP_TO::52
MARKET_ROLLUP::SHIP_TO::53
MARKET_ROLLUP::SHIP_TO::54
```

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

```
MARKET_ROLLUP::ACCOUNT::23
MARKET_ROLLUP::ACCOUNT::23
MARKET_ROLLUP::SHIP_TO::51
MARKET_ROLLUP::SHIP_TO::52
MARKET_ROLLUP::SHIP_TO::53
MARKET_ROLLUP::SHIP_TO::54
```

Using the extract Method

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

Example 7–3 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 first two are the primary `NumberSource` objects for the `UNITS` and the `UNIT_PRICE` measures, and the third is a `NumberSource` derived from a mathematical operation on the primary `NumberSource` objects.

The example selects values from hierarchies of the dimensions of the `NumberSource` for the `UNITS` measure. Two of the dimensions are the dimensions of the `NumberSource` for the `UNIT_PRICE` measure. The example produces `sales`, which is the result of the `times` method of `units` with `unitPrice` as the `rhs` parameter of the method.

Next, the example creates a list `Source`, `measDim`, which has the three `NumberSource` objects as its 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 7–3 Using the extract Method

```
Source prodSel = prodHier.selectValues(new String[]
                                       {"PRODUCT_ROLLUP::ITEM::13",
                                        "PRODUCT_ROLLUP::ITEM::14",
                                        "PRODUCT_ROLLUP::ITEM::15"});
Source chanSel = chanHier.selectValue("CHANNEL_ROLLUP::CHANNEL::2");
Source timeSel = timeHier.selectValue("CALENDAR::MONTH::59");
Source custSel = custHier.selectValue("SHIPMENTS_ROLLUP::ALL_CUSTOMERS::1");

Source sales = units.times(unitPrice);

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_ROLLUP::ALL_CUSTOMERS::1
CHANNEL_ROLLUP::CHANNEL::2
CALENDAR::MONTH::59

         UNITS SOLD   TOTAL OF UNIT PRICES   SALES AMOUNT
ITEM     ----------   --------------------   -------------
----
 13          39             2,395.63            93,429.57
 14          37             3,147.85           116,470.45
 15          26             2,993.29            77,825.54
```

Using the join Method

You use the `join` method to produce a `Source` that has the elements of its 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 7–4 and Example 7–5 demonstrate the use of two of those constants, `COMPARISON_RULE_REMOVE` and `COMPARISON_RULE_DESCENDING`. Example 7–6 also uses `COMPARISON_RULE_REMOVE`.

Example 7–4 produces a result similar to Example 7–2. It uses `mktRollup`, which is the `Source` for a hierarchy of the `CUSTOMER` dimension, and `mktRollupAncestors`, which is the `Source` for the ancestors attribute for the hierarchy. It also uses `mktRollupDescendants`, 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 `mktRollupDescendants` produces `mktRollupDescendantsOnly`, which specifies the descendants of `mktRollup`, and which has `mktRollup` as an input because the `comparison` parameter of the join operation is the `Source` that results from the `mktRollup.value()` method.

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

The next join operation performs such a match. It matches the joined `Source`, `mktRollup`, to the input of `mktRollupDescendantsOnly`, to produce `selValDescendantsOnly`, which specifies the descendants of the selected hierarchy value but does not include the selected value because `mktRollupDescendantsOnly` 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 7–4 Using COMPARISON_RULE_REMOVE

```
Source selVal = mktRollup.selectValue("MARKET_ROLLUP::ACCOUNT::23");
Source mktRollupDescendantsOnly =
      mktRollupDescendants.join(mktRollupDescendants.getDataType().value(),
                                mktRollup.value(),
                                Source.COMPARISON_RULE_REMOVE);

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

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

A `Cursor` for `selValDescendants` has the following values.

```
MARKET_ROLLUP::ACCOUNT::23
MARKET_ROLLUP::SHIP_TO::51
MARKET_ROLLUP::SHIP_TO::52
MARKET_ROLLUP::SHIP_TO::53
MARKET_ROLLUP::SHIP_TO::54
```

A `Cursor` for `selValDescendantsOnly` has the following values.

```
MARKET_ROLLUP::SHIP_TO::51
MARKET_ROLLUP::SHIP_TO::52
MARKET_ROLLUP::SHIP_TO::53
MARKET_ROLLUP::SHIP_TO::54
```

Example 7–5 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_ROLLUP` 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` 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 7–5 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
-----------------   --------------

     8,536.77       Portable PCs
     5,613.08       Desktop PCs
     1,273.00       CD-ROM
       830.74       Memory
       795.24       Monitors
       448.06       Documentation
       364.93       Accessories
       318.61       Modems/Fax
       131.84       Operating Systems
```

Using the position Method

You use the `position` method to produce a `Source` that has the positions of the elements of its base and has the base as an input. Example 7–6 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 MdmLevel 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 its element values.

The example then produces levelMembers, which is a Source that specifies the elements 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 elements of each level, by level, when its 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 elements of the levels because it removes all of the other elements 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 7–6 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.getLevels();
for (int i = 0; i < levelSources.length; i++)
{
  levelSources[i] = ((MdmLevel) 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 element in the level, and the right column is the local value of the element.

```
Level  Level Position  Level Value
-----  --------------  -----------
YEAR           1              1
YEAR           7            119
QUARTER        1              5
QUARTER       26            116
MONTH          1             19
MONTH         77            107
```

Using the recursiveJoin Method

You use the `recursiveJoin` method to produce a `Source` that has its elements ordered hierarchically. You use the `recursiveJoin` method only with the `Source` for an `MdmHierarchy` or on 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 its 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 7–7 uses a `recursiveJoin` method that lists the parents first, restricts the parents to the base, and does not add the `joined` `Source` is 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 color attribute of each element.

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

The prodColorAttr object in the second recursiveJoin method is the Source for a color attribute of the hierarchy. Only the elements of the ITEM level of the hierarchy have a related color value. Because the elements in the aggregate levels TOTAL_PRODUCT, CLASS, and FAMILY, do not have related colors, the color 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 color, so their values are NA, also.

The second recursiveJoin method joins the color 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 color 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 color attribute to produce a Source that has the color values as its 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 7–7 Sorting Products Hierarchically By Color

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

Source sortedHierNullsFirst =
 prodRollup.recursiveJoin(prodColorAttr,
                          prodColorAttr.getDataType(),
                          prodParent,
                          Source.COMPARISON_RULE_ASCENDING_NULLS_FIRST);
Source result2 = prodColorAttr.join(sortedHierNullsFirst);

Source sortedHierNullsLast =
 prodRollup.recursiveJoin(prodColorAttr,
```

```
                            prodColorAttr.getDataType(),
                            prodParent,
                            Source.COMPARISON_RULE_DESCENDING_NULLS_LAST);
Source result3 = prodColorAttr.join(sortedHierNullsLast);
```

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

```
Product Dimension Element Value
-------------------------------
PRODUCT_ROLLUP::TOTAL_PRODUCT::1
PRODUCT_ROLLUP::CLASS::2
PRODUCT_ROLLUP::FAMILY::4
PRODUCT_ROLLUP::ITEM::13
PRODUCT_ROLLUP::ITEM::14
PRODUCT_ROLLUP::ITEM::15
PRODUCT_ROLLUP::FAMILY::5
PRODUCT_ROLLUP::ITEM::16
PRODUCT_ROLLUP::ITEM::17
PRODUCT_ROLLUP::ITEM::18
...
```

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

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

```
Product Dimension Element Value   Color Value
--------------------------------  -----------
PRODUCT_ROLLUP::TOTAL_PRODUCT::1     NA
PRODUCT_ROLLUP::CLASS::2             NA
PRODUCT_ROLLUP::FAMILY::4            NA
PRODUCT_ROLLUP::ITEM::14             NA
PRODUCT_ROLLUP::ITEM::15             Black
PRODUCT_ROLLUP::ITEM::13             Silver
PRODUCT_ROLLUP::FAMILY::5            NA
PRODUCT_ROLLUP::ITEM::18             NA
PRODUCT_ROLLUP::ITEM::17             Beige
PRODUCT_ROLLUP::ITEM::16             Silver
...
```

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

```
Product Dimension Element Value   Color Value
-------------------------------   -----------
PRODUCT_ROLLUP::TOTAL_PRODUCT::1     NA
PRODUCT_ROLLUP::CLASS::2             NA
PRODUCT_ROLLUP::FAMILY::4            NA
PRODUCT_ROLLUP::ITEM::14             NA
PRODUCT_ROLLUP::ITEM::13             Silver
PRODUCT_ROLLUP::ITEM::15             Black
PRODUCT_ROLLUP::FAMILY::5            NA
PRODUCT_ROLLUP::ITEM::18             NA
PRODUCT_ROLLUP::ITEM::16             Silver
PRODUCT_ROLLUP::ITEM::17             Beige
...
```

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 7–8 demonstrates the selection of such a subset. In the example, `shipRollup` is a `Source` for the `SHIPMENTS_ROLLUP` hierarchy of the `CUSTOMER` dimension. The `selectValues` method of `shipRollup` produces `custSel`, which is a selection of some of the elements of `shipRollup`. 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 7–8 Selecting a Subset of the Elements of a Source

```
StringSource custSel = (StringSource) shipRollup.selectValues(new String[]
                                          {"SHIPMENTS_ROLLUP::SHIP_TO::60",
                                           "SHIPMENTS_ROLLUP::SHIP_TO::61",
                                           "SHIPMENTS_ROLLUP::SHIP_TO::62",
                                           "SHIPMENTS_ROLLUP::SHIP_TO::63"});

Source custSel2 = custSel.selectValues(new String[]
                                       {"SHIPMENTS_ROLLUP::SHIP_TO::60",
                                        "SHIPMENTS_ROLLUP::SHIP_TO::62"});

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_ROLLUP::SHIP_TO::60   SHIPMENTS_ROLLUP::SHIP_TO::60
SHIPMENTS_ROLLUP::SHIP_TO::60   SHIPMENTS_ROLLUP::SHIP_TO::61
SHIPMENTS_ROLLUP::SHIP_TO::60   SHIPMENTS_ROLLUP::SHIP_TO::62
SHIPMENTS_ROLLUP::SHIP_TO::60   SHIPMENTS_ROLLUP::SHIP_TO::63
SHIPMENTS_ROLLUP::SHIP_TO::62   SHIPMENTS_ROLLUP::SHIP_TO::60
SHIPMENTS_ROLLUP::SHIP_TO::62   SHIPMENTS_ROLLUP::SHIP_TO::61
SHIPMENTS_ROLLUP::SHIP_TO::62   SHIPMENTS_ROLLUP::SHIP_TO::62
SHIPMENTS_ROLLUP::SHIP_TO::62   SHIPMENTS_ROLLUP::SHIP_TO::63
```

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_ROLLUP::SHIP_TO::60   SHIPMENTS_ROLLUP::SHIP_TO::60
SHIPMENTS_ROLLUP::SHIP_TO::62   SHIPMENTS_ROLLUP::SHIP_TO::62
```

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.

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 7–9 creates a cube that has the quantity of units sold as its data. 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 prepares and commits the current `Transaction`, and then creates a `Cursor` for the query and displays its 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 prepares and commits the `Transaction`, creates another `Cursor` for the query, and displays its 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 `context` object has a method that displays the values of the `Cursor` in a crosstab format.

Example 7–9 Creating a Cube and Pivoting Its Edges

```
// Create Parameter objects with values from the default hierarchies
// of the CUSTOMER and PRODUCT dimensions.
StringParameter custParam =
                     new StringParameter(dp, "SHIPMENTS_ROLLUP::REGION::9");
StringParameter prodParam =
                     new StringParameter(dp, "PRODUCT_ROLLUP::FAMILY::4");

// Create parameterized Source objects using the Parameter objects.
StringSource custParamSrc = dp.createParameterizedSource(custParam);
StringSource prodParamSrc = dp.createParameterizedSource(prodParam);

// 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::2"
                                             "CALENDAR::YEAR::3",
                                             "CALENDAR::YEAR::4"});
Source chanSel = chanHier.selectValues(new String[]
                                      {"CHANNEL_ROLLUP::CHANNEL::2",
                                       "CHANNEL_ROLLUP::CHANNEL::3",
                                       "CHANNEL_ROLLUP::CHANNEL::4"});

// 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);

// Get the TransactionProvider.
TransactionProvider tp = context.getTransactionProvider();
// Prepare and commit the currentTransaction.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  context.println("Cannot prepare the current Transaction. " + e
}
tp.commitCurrentTransaction();

// Create a Cursor for the query.
CursorManagerSpecification cMngrSpec =
                        dp.createCursorManagerSpecification(cube);
SpecifiedCursorManager  spCMngr = dp.createCursorManager(cMngrSpec);
Cursor cubeCursor = spCMngr.createCursor();
// Display the values of the Cursor as a crosstab.
context.displayCursorAsCrosstab(cubeCursor);

// Change the customer parameter value.
custParam.setValue("SHIPMENTS_ROLLUP::REGION::10");

// Reset the Cursor position to 1 and display its values again.
cubeCursor.setPosition(1);
context.println(" ");
context.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);

// Prepare and commit the current Transaction.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  context.println("Cannot prepare the current Transaction. " + e);
}
tp.commitCurrentTransaction();

// Create another Cursor.
cMngrSpec = dp.createCursorManagerSpecification(cube);
spCMngr = dp.createCursorManager(cMngrSpec);
cubeCursor = spCMngr.createCursor();
context.displayCursorAsCrosstab(cubeCursor);

// Change the product parameter value.
prodParam.setValue("PRODUCT_ROLLUP::FAMILY::5");

// Reset the Cursor position to 1
cubeCursor.setPosition(1);
context.println(" ");
context.displayCursorAsCrosstab(cubeCursor);
```

The following crosstab has the values of `cubeCursor` displayed by the first `displayCursorAsCrosstab` method.

```
Portable PCs
Europe

        Direct Sales  Catalog  Internet
1999              86     1986         0
2000             193     1777        10
2001             196     1449       215
```

The following crosstab has the values of `cubeCursor` after the example changed the value of the `custParam` `Parameter` object.

```
Portable PCs
North America

        Direct Sales  Catalog  Internet
1999             385     6841         0
2000             622     6457        35
2001             696     5472       846
```

The next crosstab has the values of `cubeCursor` after pivoting the column and row edges.

```
Portable PCs
North America

                1999    2000    2001
Direct Sales     385     622     696
Catalog         6841    6457    5472
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
Direct Sales     793    1224    1319
Catalog        14057    1321   11337
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 7–10 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_ROLLUP` hierarchy of the `PRODUCT` dimension.
- `prodRollup`, which is the `Source` for the `PRODUCT_ROLLUP` hierarchy.
- `prodRollupParentAttr`, which is the `Source` for the parent attribute of the hierarchy.

- `prodRollupAncsAttr`, 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.
- `context`, which has methods that prepare and commit the current `Transaction`, that create a `Cursor` for a `Source`, that display text, and that display the values of the `Cursor`.

Example 7–10 Drilling in a Hierarchy

```
int pos = 2;
// 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 = prodRollup.join(prodRollup.value(), levelElement);

// Get ancestors of the level element.
Source levelElementAncs = prodRollupAncsAttr.join(prodRollup,
                                                  levelElement);
// Get the parent of the level element.
Source levelElementParent = prodRollupParentAttr.join(prodRollup,
                                                      levelElement);
// Get the children of a parent.
Source prodRollupChildren = prodRollup.join(prodRollupParentAttr,
                                            prodRollup.value());

// Select the children of the level element.
Source levelElementChildren = prodRollupChildren.join(prodRollup,
                                                      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(prodRollup, levelElementParent, true);
```

```
// Get the short value descriptions the ancestors.
Source levelElementAncsWithShortDescr =
            prodShortLabel.join(prodRollup, levelElementAncs, true);

// Prepare and commit the current Transaction.
context.commit();

// Create Cursor objects and display their values.
context.println("Level element values:");
context.displayResult(levelSrcWithShortDescr);
context.println("\nLevel element at position " + pos + ":");
context.displayResult(levelElement);
context.println("\nParent of the level element:");
context.displayResult(levelElementParent);
context.println("\nChildren of the level element:");
context.displayResult(levelElementChildrenWithShortDescr);
context.println("\nAncestors of the level element:");
context.displayResult(levelElementAncs);
```

The following list has the values of the `Cursor` objects created by the `displayResults` methods.

```
Level element values:

1: (PRODUCT_ROLLUP::FAMILY::4,Portable PCs)
2: (PRODUCT_ROLLUP::FAMILY::5,Desktop PCs)
3: (PRODUCT_ROLLUP::FAMILY::6,Operating Systems)
4: (PRODUCT_ROLLUP::FAMILY::7,Accessories)
5: (PRODUCT_ROLLUP::FAMILY::8,Monitors)
6: (PRODUCT_ROLLUP::FAMILY::9,Modems/Fax)
7: (PRODUCT_ROLLUP::FAMILY::10,Memory)
8: (PRODUCT_ROLLUP::FAMILY::11,CD-ROM)
9: (PRODUCT_ROLLUP::FAMILY::12,Documentation)

Level element at position 2:

1: PRODUCT_ROLLUP::FAMILY::5

Parent of the level element:

1: (PRODUCT_ROLLUP::CLASS::2,Hardware)
```

```
Children of the level element:

1: (PRODUCT_ROLLUP::ITEM::16,Sentinel Standard)
2: (PRODUCT_ROLLUP::ITEM::17,Sentinel Financial)
3: (PRODUCT_ROLLUP::ITEM::18,Sentinel Multimedia)

Ancestors of the level element:

1: (PRODUCT_ROLLUP::TOTAL_PRODUCT::1,Total Product)
2: (PRODUCT_ROLLUP::CLASS::2,Hardware)
3: (PRODUCT_ROLLUP::FAMILY::5,Desktop PCs)
```

Sorting Hierarchically by Measure Values

Example 7–11 uses the `recursiveJoin` method to sort the elements of the `PRODUCT_ROLLUP` 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 its element values and `sortedProductsShortDescr` as its 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.

- `prodRollup`, which is the `Source` for the `PRODUCT_ROLLUP` 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_ROLLUP` hierarchy.
- `prodRollupAncsAttr`, which is the `Source` for the ancestors attribute of the 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. Its value is `SHIPMENTS_ROLLUP::ALL_CUSTOMERS::1`, which is all customers.

- `chanSel`, which is a `Source` that specifies a single element of the default hierarchy of the `CHANNEL` dimension. Its value is `CHANNEL_ROLLUP::CHANNEL::2`, which is the direct sales channel.
- `timeSel`, which is a `Source` that specifies a single element of the default hierarchy of the `TIME` dimension. Its value is `CALENDAR::YEAR::4`, which is the year 2001.

Example 7–11 Hierarchical Sorting by Measure Value

```
Source sortedProduct =
         prodRollup.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_ROLLUP` 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 ten 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 Monitors 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  1 Total Product                    43,783
        CLASS  2 Hardware                         16,541
       FAMILY  4 Portable PCs                      1,192
         ITEM 15 Envoy Ambassador                    330
```

```
  ITEM 14 Envoy Executive                    385
  ITEM 13 Envoy Standard                     477
FAMILY  8 Monitors                         1,193
  ITEM 21 Monitor- 19 Super VGA              207
  ITEM 20 Monitor- 15 Super VGA              986
  ...
 CLASS 3 Software/Other                   27,242
  ...
FAMILY  7 Accessories                      18,949
  ITEM 22 Envoy External Keyboard            146
  ITEM 23 External 101-key keyboard          678
  ITEM 32 Multimedia speakers- 5 cones       717
  ITEM 46 Standard Mouse                     868
  ITEM 27 Multimedia speakers- 3 cones      1,120
  ITEM 31 1.44MB External 3.5 Diskette      1,145
  ITEM 48 Keyboard Wrist Rest               2,231
  ITEM 19 Laptop carrying case              3,704
  ITEM 47 Deluxe Mouse                      3,884
  ITEM 30 Mouse Pad                         4,456
```

Using NumberSource Methods To Compute the Share of Units Sold

Example 7–12 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 `prodRollup`, which is the `Source` for a hierarchy of the `PRODUCT` dimension, to produce `allProds`, which specifies a single element with the value `PRODUCT_ROLLUP::TOTAL_PRODUCT::1`, 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 `allProds` 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 its inputs.

The `prodFamilies` object is the `Source` for the `FAMILY` level of the `PRODUCT_ROLLUP` 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 7–12 Getting the Share of Units Sold

```
Source allProds = prodRollup.selectValue("PRODUCT_ROLLUP::TOTAL_PRODUCT::1");
NumberSource totalUnits = (NumberSource) units.joinHidden(allProds);
Source productShare = units.div(totalUnits).times(100);
Source result =
      productShare.join(prodFamilies)
                  .join(timeHier, "CALENDAR::YEAR::4")
                  .join(chanHier, "CHANNEL_ROLLUP::CHANNEL::2")
                  .join(custHier, "SHIPMENTS_ROLLUP::ALL_CUSTOMERS::1");
```

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 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_ROLLUP::FAMILY::4             2.72%
PRODUCT_ROLLUP::FAMILY::5             5.13%
PRODUCT_ROLLUP::FAMILY::6            12.54%
PRODUCT_ROLLUP::FAMILY::7            43.28%
PRODUCT_ROLLUP::FAMILY::8             2.73%
PRODUCT_ROLLUP::FAMILY::9            11.92%
PRODUCT_ROLLUP::FAMILY::10            3.57%
PRODUCT_ROLLUP::FAMILY::11           11.71%
PRODUCT_ROLLUP::FAMILY::12            6.4%
```

Ranking Dimension Elements by Measure Value

Example 7–13 produces two results. The first is `result1`, which is a `Source` that specifies the rank of two families of products and their members in the order of the sales of all product units. The second is `result2`, which ranks those families and their members by quantity of units sold compared to each other.

The `units` object is the `Source` for the `UNITS` measure, and `prodRollup` is the `Source` for the `PRODUCT_ROLLUP` hierarchy of the `PRODUCT` dimension. The `join` method of `units` produces a `Source` that specifies units sold values for each element of the hierarchy.

The `select` method has as its `filter` parameter the `BooleanSource` produced by the `gt` method of the `Source` that results from the `units.value()` method. The `Source` that results from the `select` method has `units` as an input. When a join operation matches a `Source` to that input, it produces a `Source` that, for each element of the `units` measure, has the Boolean value `true` for every units sold value that is greater than the current element value.

The `count` method then produces a `Source` that has, for each element of the measure, the total number of all the products that have greater sales quantities. The product element with the greatest quantity of units sold therefore has a count of zero. The `plus` method then adds 1 to each count amount so that the rank values begin with the number 1.

The `join` method of the `Source` produced by the `plus` method selects the elements of `pcParentsAndChildren` from all of the elements of the product hierarchy. The `joinHidden` methods then match `Source` objects that specify selections of the dimensions that are the remaining inputs of the `units` measure to produce `result1`, which specifies the calculation of the rank of the selected product elements relative to all of the product elements for the customer, time, and channel values.

The methods that product `result2` are the same except that the first `join` produces a `Source` that specifies the `units` elements only for the elements of `pcParentsAndChildren`. The `select`, `gt`, `count`, and `plus` methods operate on only those selected elements of the hierarchy. The `result2 Source` therefore specifies the calculation of the rank of the selected product elements relative to each other rather than relative to all product elements.

Example 7–13 Ranking Products by Units Sold

```
// First result:  PC products unit sales ranked relative to all products.
Source result1 = units.join(prodRollup,
                            dp.getEmptySource(),
                            Source.COMPARISON_RULE_REMOVE,
                            false)
                      .select(units.value().gt(units)).count().plus(1)
                      .join(pcParentsAndChildren)
                      .joinHidden(custSel)
                      .joinHidden(timeSel)
                      .joinHidden(chanSel);

// Second result:  PC products unit sales ranked relative to each other.
Source result2 = units.join(pcParentsAndChildren,
                            dp.getEmptySource(),
                            Source.COMPARISON_RULE_REMOVE,
                            false)
                      .select(units.value().gt(units)).count().plus(1)
                      .join(prodRollup)
                      .joinHidden(custSel)
                      .joinHidden(timeSel)
                      .joinHidden(chanSel);
```

A `Cursor` for the `result1 Source` has the following values, displayed in a table with column headings and formatting added. The left column has the product element value and the right column has the rank of that product compared to all product units sold.

Product Element	Rank Compared To Total Products Sold
PRODUCT_ROLLUP::FAMILY::5	16
PRODUCT_ROLLUP::FAMILY::4	21
PRODUCT_ROLLUP::ITEM::16	29
PRODUCT_ROLLUP::ITEM::17	31
PRODUCT_ROLLUP::ITEM::18	34
PRODUCT_ROLLUP::ITEM::13	35
PRODUCT_ROLLUP::ITEM::14	37
PRODUCT_ROLLUP::ITEM::15	39

A `Cursor` for the `result2 Source` has the following values, displayed in a table with column headings and formatting added. The left column has the product

element value and the right column has the rank of that product compared to the other product family members.

```
                                Rank Compared To
Product Element                    Each Other
-------------------------       ---------------
PRODUCT_ROLLUP::FAMILY::5              1
PRODUCT_ROLLUP::FAMILY::4              2
PRODUCT_ROLLUP::ITEM::16               3
PRODUCT_ROLLUP::ITEM::17               4
PRODUCT_ROLLUP::ITEM::18               5
PRODUCT_ROLLUP::ITEM::13               6
PRODUCT_ROLLUP::ITEM::14               7
PRODUCT_ROLLUP::ITEM::15               8
```

Selecting Based on Time Series Operations

This section has two examples of using methods that operate on a series of time dimension elements. Example 7–14 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 dimension level.

In the example, `dp` is the `DataProvider`. Its `createListSource` method creates `measuresDim`, which has the `unitPrice` and `unitPriceLag4` `Source` objects as its 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` 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_ROLLUP::FAMILY::5`, 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 7–14 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 left 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` hierarchy. The table includes only the first eight quarters.

Quarter	Unit Price	Unit Price Four Quarters Before
CALENDAR::QUARTER::5	16125.24	NA
CALENDAR::QUARTER::6	16226.89	NA
CALENDAR::QUARTER::7	16039.61	NA
CALENDAR::QUARTER::8	15526.53	NA
CALENDAR::QUARTER::9	21553.14	16,125.24
CALENDAR::QUARTER::10	21034.61	162,26.89
CALENDAR::QUARTER::11	21135.51	16,039.61
CALENDAR::QUARTER::12	19600.98	15,526.53

...

Example 7–15 uses the same `unitPrice`, `quarterLevel`, and `prodSel` objects as Example 7–14, 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 its `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 7–15 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.

Quarter	Unit Price	Unit Price Moving Total Current Plus Next Three Periods
CALENDAR::QUARTER::5	16,125.24	63,918.27
CALENDAR::QUARTER::6	16,226.89	69,346.17
CALENDAR::QUARTER::7	16,039.61	74,153.89
CALENDAR::QUARTER::8	15,526.53	79,249.79
CALENDAR::QUARTER::9	21,553.14	80,206.84
CALENDAR::QUARTER::10	21,034.61	80,206.84
CALENDAR::QUARTER::11	21,135.51	77,638.28
...		

Selecting a Set of Elements Using Parameterized Source Objects

Example 7–16 uses `NumberParameter` objects to create parameterized `Source` objects. Those objects are the `bottom` and `top` parameters for the `interval` method of `prodRollup`. That method produces `paramProdSelInterval`, which is a `Source` that specifies the set of elements of `prodRollup` 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.
- prodRollup, which is the Source for the PRODUCT_ROLLUP 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.
- chanRollup, which is the Source for the CHANNEL_ROLLUP hierarchy of the CHANNEL dimension.
- calendar, which is the Source for the CALENDAR hierarchy of the TIME dimension.
- shipRollup, which is the Source for the SHIPMENTS_ROLLUP hierarchy of the CUSTOMER dimension.
- context, which has methods that prepare and 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 its 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 7–16 Selecting a Range With NumberParameter Objects

```
NumberParameter startParam = new NumberParameter(dp, 1);
NumberParameter endParam = new NumberParameter(dp, 6);

NumberSource startParamSrc = dp.createParameterizedSource(startParam);
NumberSource endParamSrc = dp.createParameterizedSource(endParam);

Source paramProdSelInterval = prodRollup.interval(startParamSrc,
                                                  endParamSrc);
Source paramProdSelIntervalShortDescr =
                                    prodShortDescr.join(paramProdSelInterval);
```

```
NumberSource result = (NumberSource)
                       units.join(chanRollup, "CHANNEL_ROLLUP::CHANNEL::4")
                            .join(calendar, "CALENDAR::YEAR::4")
                            .join(shipRollup,
                                  "SHIPMENTS_ROLLUP::ALL_CUSTOMERS::1")
                            .join(paramProdSelIntervalShortDescr);

// Get the TransactionProvider and prepare and commit the
// current transaction. These operations are not shown.

CursorManagerSpecification cMngrSpec =
                        dp.createCursorManagerSpecification(results);
SpecifiedCursorManager  spCMngr = dp.createCursorManager(cMngrSpec);
Cursor resultCursor = spCMngr.createCursor();

context.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.
context.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_ROLLUP::TOTAL_PRODUCT::1 Total Product            55,872
PRODUCT_ROLLUP::CLASS::2         Hardware                 21,301
PRODUCT_ROLLUP::FAMILY::4        Portable PCs              1,420
PRODUCT_ROLLUP::ITEM::13         Envoy Standard              550
PRODUCT_ROLLUP::ITEM::14         Envoy Executive             482
PRODUCT_ROLLUP::ITEM::15         Envoy Ambassador            388
```

```
PRODUCT_ROLLUP::FAMILY::5        Desktop PCs              2,982
PRODUCT_ROLLUP::ITEM::16         Sentinel Standard        1,092
PRODUCT_ROLLUP::ITEM::17         Sentinel Financial       1,015
PRODUCT_ROLLUP::ITEM::18         Sentinel Multimedia        875
PRODUCT_ROLLUP::FAMILY::8        Monitors                 1,505
PRODUCT_ROLLUP::ITEM::20         Monitor- 15 Super VGA    1,238
```

8

Using a TransactionProvider

This chapter describes the Oracle OLAP API `Transaction` and `TransactionProvider` interfaces and describes how you use implementations of those interfaces in an application. You must create a `TransactionProvider` before you can create a `DataProvider`, and you must use methods of the `TransactionProvider` to prepare and commit a `Transaction` before you can create a `Cursor` for a derived `Source`.

This chapter includes the following topics:

For the complete code for most of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

About Creating a Query in a Transaction

The Oracle OLAP API is transactional. Each step in creating a query occurs in the context of a `Transaction`. One of the first actions of an OLAP API application is to create a `TransactionProvider`. The `TransactionProvider` provides `Transaction` objects to the application.

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, its effects 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 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 10.
- 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 6. For information on Template objects, see Chapter 11.

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, 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.

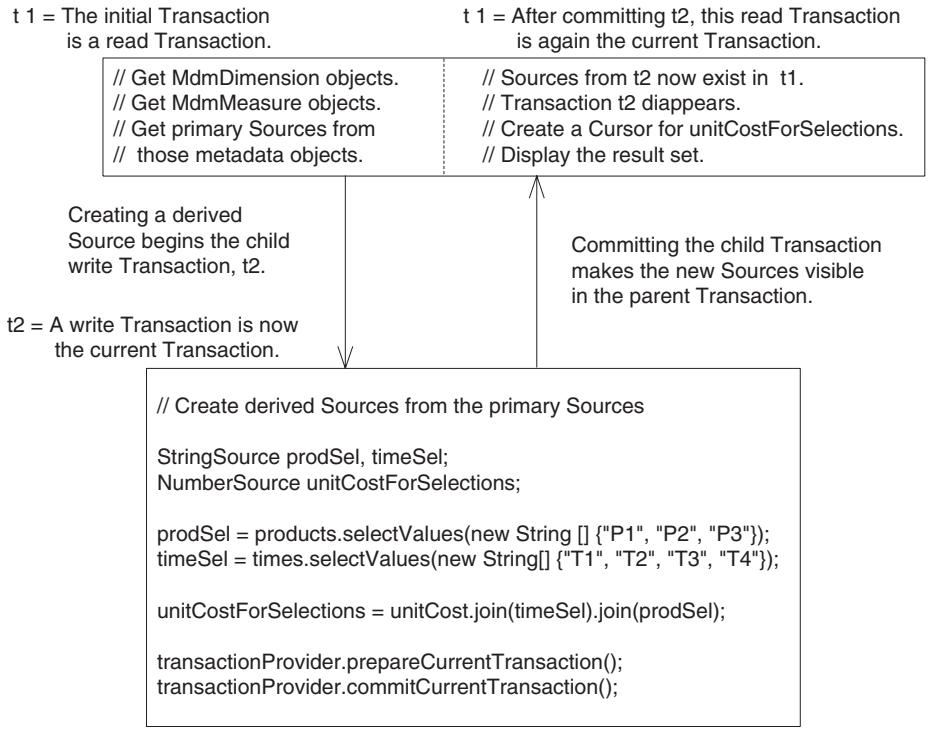
Before you can create a `Cursor` to fetch the result set specified by a derived `Source`, you must move the `Source` from the child write `Transaction` into the parent read `Transaction`. To do so, you prepare and commit the `Transaction`.

Preparing and Committing a Transaction

To move a `Source` that you created in a child `Transaction` into the parent read `Transaction`, call the `prepareCurrentTransaction` and `commitCurrentTransaction` methods 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.

The following figure illustrates the process of moving a `Source` created in a child write `Transaction` into its parent read `Transaction`.

Figure 8–1 Committing a Write Transaction into Its Parent Read Transaction



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.

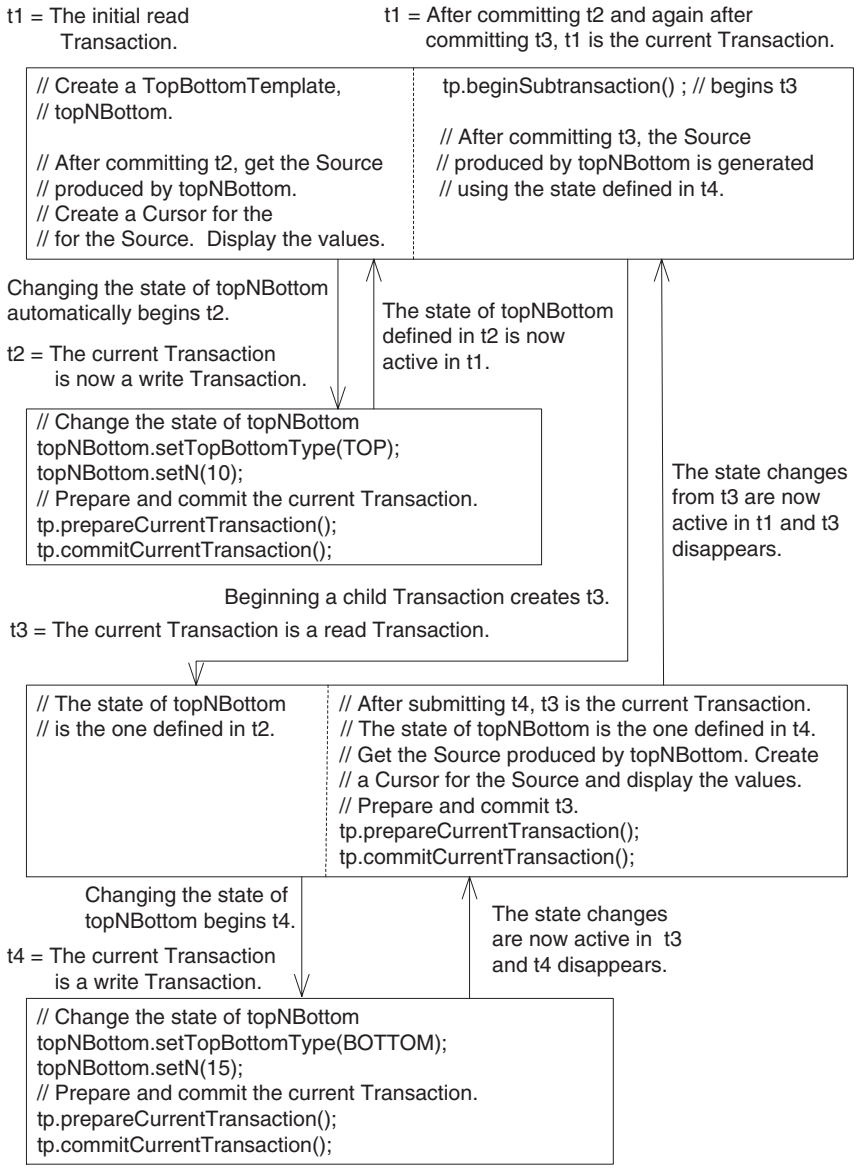
Beginning a Child Transaction

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 prepare and commit the write `Transaction` into its 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 prepare and commit the child read `Transaction` into the parent read `Transaction`.

The following figure illustrates beginning a child read `Transaction`, creating `Source` objects in a write `Transaction`, and committing the write `Transaction` into its parent read `Transaction`. The figure then shows committing the child read `Transaction` into its parent read `Transaction`. In the figure, `tp` is the `TransactionProvider`.

Figure 8–2 Committing a Child Read Transaction into Its Parent 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 8–2.

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 prepare and commit that Transaction. Likewise, once you commit a Transaction, you cannot roll it back.

Example 8–1 Rolling Back a Transaction

The following example creates a TopBottomTemplate and sets its state. The example begins a child Transaction that sets a different state for the TopBottomTemplate and then rolls back the child Transaction. The TransactionProvider is tp.

```
// The current Transaction is a read Transaction, t1.
// Create a TopBottomTemplate using product as the base
// and dp as the DataProvider.
TopBottomTemplate topNBottom = new TopBottomTemplate(product, 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());

// Prepare and commit the Transaction t2.
try
{
  tp.prepareCurrentTransaction();
}
```

```
catch(NotCommittableException e)
{
  context.println("Cannot commit the Transaction. " + e);
}
tp.commitCurrentTransaction();           //t2 disappears

// The current Transaction is now t1.
// Create a Cursor and display the results (operations not shown).

// 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);

// Prepare and commit the Transaction.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  context.println("Cannot commit the Transaction. " + e);
}
tp.commitCurrentTransaction();           // t4 disappears

// Create a Cursor and display the results. // t3 is the current Transaction
// Close 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.
```

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 = getCurrentTransaction();
```

To make a previously saved Transaction the current Transaction, you call the setCurrentTransaction method of the TransactionProvider, as in the following example.

```
setCurrentTransaction(t1);
```

Using TransactionProvider Objects

In the Oracle OLAP API, the TransactionProvider interface is implemented by the ExpressTransactionProvider concrete class. Before you create a DataProvider, you must create a new instance of an ExpressTransactionProvider. You then pass that TransactionProvider to the DataProvider constructor. The TransactionProvider provides Transaction objects to your application.

As described in "Preparing and Committing a Transaction", you use the prepareCurrentTransaction and commitCurrentTransaction methods 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, you must close the CursorManager before rolling back the Transaction.

To demonstrate how to use Transaction objects to modify dynamic queries, Example 8–2 builds on the TopBottomTest application defined in Chapter 11. To help track the Transaction objects, the example saves the different Transaction objects with calls to the getCurrentTransaction method. In the example, tp object is the TransactionProvider, and context is an object that has methods that create Cursor objects and display their values.

Example 8–2 replaces the lines of the code from the `run` method of Example 11–4, which is the `TopBottomTest` class, from the following comment in Example 11–4 to the end of the `TopBottomTest.run` method.

```
// Replace from here for the Using Child Transaction example.
```

Example 8–2 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();

// 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();

// Prepare and commit the write Transaction writeT3.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
 context.println("Cannot commit current Transaction. " + e);
}
tp.commitCurrentTransaction();

// 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.
context.displayResult(topNBottom.getSource());

// 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();

// Prepare and commit writeT5.
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  context.println("Cannot commit current Transaction. " + e);
}
tp.commitCurrentTransaction();

// 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.
context.displayResult(topNBottom.getSource();

// Commit the grandchild into the child.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  context.println("Cannot commit current Transaction. " + e);
}
tp.commitCurrentTransaction();

// 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.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
 context.println("Cannot commit the child Transaction. " + e);
}
tp.commitCurrentTransaction();

// Transaction parentT1 is now the current Transaction. Again,
// you can roll back the childT2 Transaction instead of
// preparing and committing it. If you do so, then the changes
// 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.
```

9

Understanding Cursor Classes and Concepts

This chapter describes the Oracle OLAP 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 10, "Retrieving Query Results".

This chapter includes the following topics:

- Overview of the OLAP API Cursor Objects
- Cursor Classes
- CursorManagerSpecification Class
- CursorInfoSpecification Classes
- CursorManager Classes
- Other Classes
- About Cursor Positions and Extent
- About Fetch Sizes

For the complete code of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

Overview of the OLAP API Cursor Objects

A `Cursor` retrieves the result set defined by a `Source`. You can also 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 ExpressSQLCursorManager by using the createSQLCursorManager method of a DataProvider. You can then use classes outside of the OLAP API to retrieve data using the generated SQL.

The Oracle OLAP API has two paths to the creation of a Cursor for a Source. The older method requires creating a CursorManagerSpecification, then creating a CursorManager, and then creating a Cursor. The newer method eliminates the CursorManagerSpecification. Instead, you simply create a CursorManager for the Source and then create a Cursor.

Creating a Cursor Using a CursorManagerSpecification

In the older method, after creating a Source that defines the data that you want to retrieve from the data store, you create a Cursor for that Source by doing the following:

1. Creating a CursorManagerSpecification by passing the Source to the createCursorManagerSpecification method of the DataProvider that you are using. The CursorManagerSpecification has CursorSpecification objects in a structure that mirrors the structure of the Source.
2. Creating a CursorManager by calling the createCursorManager method of the DataProvider and passing it the CursorManagerSpecification. The CursorManager creates Cursor objects. It also manages the local data cache for its Cursor objects and is aware of changes to the Source for a dynamic query or a parameterized Source. If the Source for the CursorManagerSpecification has inputs, then you must also pass to the createCursorManager method an array of Source objects for those inputs.
3. Creating a Cursor by calling the createCursor method of the CursorManager. The structure of the Cursor mirrors the structures of the CursorManagerSpecification and the Source. The CursorSpecification objects of a CursorManagerSpecification specify the behavior of their corresponding Cursor objects. If the Source for the CursorManagerSpecification has inputs, then you must also pass to the createCursor method an array of CursorInput objects that specify values for the input Source objects.

For an example of creating a Cursor using this method, see Chapter 10.

This architecture provides great flexibility in fetching data from a result set and in selecting data to display. You can do the following:

- Create more than one CursorManagerSpecification object for the same Source. You can specify different behavior on the CursorSpecification components of the various CursorManagerSpecification objects in order to retrieve and display different sets of values from the same result set. You might want to do this when displaying the data from a Source in different formats, such as in a table and a crosstab.
- Receive notification that the Source produced by the Template has changed. If you add a CursorManagerUpdateListener to the CursorManager for a Source, then the CursorManager notifies the CursorManagerUpdateListener when the Source for a dynamic query has changed and you that therefore need to update the CursorManagerSpecification for the CursorManager.
- Update the CursorManagerSpecification for a CursorManager. If you are using Template objects to produce a dynamic query and the state of a Template changes, then the Source produced by the Template changes. If you have created a Cursor for the Source produced by the Template, then you need to replace the CursorManagerSpecification for the CursorManager with an updated CursorManagerSpecification for the changed Source. You can then create a new Cursor from the CursorManager.
- Create different Cursor objects from the same CursorManager and set different fetch sizes on those Cursor objects. You might do this when you want to display the same data as a table and as a graph.

This older method of creating a CursorManager returns an ExpressSpecifiedCursorManager.

Creating a Cursor Without a CursorManagerSpecification

In the newer method, 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 its methods 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.

This newer method of creating a CursorManager returns an ExpressDataCursorManager.

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.

- 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.

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 prepareCurrentTransaction and commitCurrentTransaction methods of the TransactionProvider your application is using. For information on Transaction and TransactionProvider objects, see Chapter 8, "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 prepare and commit the Transaction again. For information on parameterized Source objects, see Chapter 6, "Understanding Source Objects".

Cursor Classes

In the `oracle.olapi.data.cursor` package, the Oracle OLAP 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 its `Source` and an output child `Cursor` for each output of the `Source`.

Structure of a Cursor

The structure of a `Cursor` mirrors the structure of its `Source`. 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 9–1 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 9–1 Creating the querySource Query

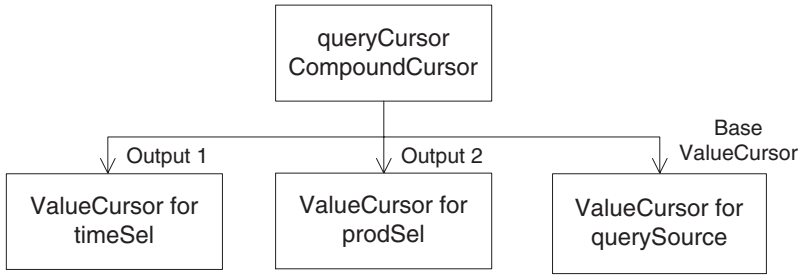
```
Source timeSel = timeHier.selectValues(new String[]
                                       {"CALENDAR::MONTH::55",
                                        "CALENDAR::MONTH::58",
                                        "CALENDAR::MONTH::61",
                                        "CALENDAR::MONTH::64"});

Source prodSel = prodHier.selectValues(new String[]
                                       {"PRODUCT_ROLLUP::ITEM::13",
                                        "PRODUCT_ROLLUP::ITEM::14",
                                        "PRODUCT_ROLLUP::ITEM::15"});

Source querySource = unitPrice.join(prodSel).join(timeSel);
```

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 10–1 in Chapter 10, 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 9–1 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 9–1 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
55	13	2426.07
55	14	3223.28
55	15	3042.22
58	13	2412.42
58	14	3107.65
58	15	3026.12
61	13	2505.57
61	14	3155.91
61	15	2892.18
64	13	2337.30
64	14	3105.53
64	15	2856.86

For examples of getting the values from a `ValueCursor`, see Chapter 10.

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 its parent, then you can successfully use the following `Cursor` methods:

- `getExtent`
- `getFetchSize`
- `getParentEnd`
- `getParentStart`
- `setFetchSize`

For examples of specifying `Cursor` behavior, see Chapter 10. For information on fetch sizes, see "About Fetch Sizes" on page 9-25. For information on the extent of a `Cursor`, see "What is the Extent of a Cursor?" on page 9-23. 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 9-21.

CursorManagerSpecification Class

A CursorManagerSpecification for a Source has one or more CursorSpecification objects. The structure of those objects reflects the structure of the Source. For example, a Source that has outputs has a top-level, or *root*, CursorSpecification for the Source, a child CursorSpecification for the values of the Source, and a child CursorSpecification for each output of the Source.

A Source that does not have any outputs has only one set of values. A CursorManagerSpecification for that Source therefore has only one CursorSpecification. That CursorSpecification is the root CursorSpecification of the CursorManagerSpecification.

You can create a CursorManagerSpecification for a multidimensional Source that has one or more inputs. If you do so, then you need to supply a Source for each input when you create a CursorManager for the CursorManagerSpecification. You must also supply a CursorInput for each input Source when you create a Cursor from the CursorManager. You might create a CursorManagerSpecification for a Source with inputs if you want to use a CursorManager to create a series of Cursor objects with each Cursor retrieving data specified by a different set of single values for the input Source objects.

The structure of a Cursor reflects the structure of its CursorManagerSpecification. A Cursor can be a single ValueCursor, for a Source with no outputs, or a CompoundCursor with child Cursor objects, for a Source with outputs. Each Cursor corresponds to a CursorSpecification in the CursorManagerSpecification. You use CursorSpecification methods to specify aspects of the behavior of the corresponding Cursor.

If your application uses Template objects, and a change occurs in the state of a Template so that the structure of the Source produced by the Template changes, then any CursorManagerSpecification objects that the application created for the Source expire. If a CursorManagerSpecification expires, then you must create a new CursorManagerSpecification. You can then either use the new CursorManagerSpecification to replace the old CursorManagerSpecification of a CursorManager or use it to create a new CursorManager. You can discover whether a CursorManagerSpecification has expired by calling the isExpired method of the CursorManagerSpecification.

CursorInfoSpecification Classes

The CursorInfoSpecification interface and its 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 CursorInfoSpecification for a Source directly by calling the createCursorInfoSpecification method of a DataProvider. You can use the methods of the CursorInfoSpecification to specify aspects of the behavior of a Cursor. You can then use the CursorInfoSpecification in creating a CursorManager by passing it as the cursorInfoSpec argument to the createCursorManager method of a DataProvider.

You can create a CursorInfoSpecification for a Source indirectly by creating a CursorManagerSpecification. You pass a Source to the createCursorManagerSpecification method of a DataProvider and the CursorManagerSpecification returned has a root CursorSpecification for that Source. If the Source has outputs, then the CursorManagerSpecification also has a child CursorSpecification for the values of the Source and one for each output of the Source.

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.
- On a CompoundCursorSpecification, specify that the default fetch size is set on the children of 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 its 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 its parent is specified.
- Accept a `CursorSpecificationVisitor`.

For more information, see "About Cursor Positions and Extent" on page 9-15 and "About Fetch Sizes" on page 9-25.

In the `oracle.olapi.data.source` package, the Oracle OLAP 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 its `CursorManagerSpecification`. For every `ValueCursorSpecification` or `CompoundCursorSpecification` of a `CursorManagerSpecification`, a `Cursor` 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 Classes

The OLAP API has the following concrete classes for creating a `Cursor` for a `Source` or for getting the SQL generated by a `Source`.

- `ExpressDataCursorManager`
- `ExpressSpecifiedCursorManager`
- `ExpressSQLCursorManager`

An `ExpressSQLCursorManager` has methods that return the SQL generated by the Oracle OLAP SQL generator for the `Source`. You create one or more `ExpressSQLCursorManager` objects by calling the `createSQLCursorManager` or `createSQLCursorManagers` methods of a `DataProvider`. You do not use an `ExpressSQLCursorManager` to create a `Cursor` to retrieve the result set of the query specified by the `Source`. Instead, you use the SQL returned by the `ExpressSQLCursorManager` with classes outside of the OLAP API to retrieve the data specified by the query.

An `ExpressDataCursorManager` or `ExpressSpecifiedCursorManager` 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 cursor manager, 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 cursor manager 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 cursor manager.

An `ExpressSpecifiedCursorManager` implements the `SpecifiedCursorManager` interface, which extends the `CursorManager` interface. A `CursorManager` has methods for creating a `Cursor`, for discovering whether the `CursorManagerSpecification` for the `CursorManager` needs updating, and for adding or removing a `CursorManagerUpdateListener`. The `SpecifiedCursorManager` interface adds methods for updating the `CursorManagerSpecification`, for discovering whether the `SpecifiedCursorManager` is open, and for closing it. Some of the `createCursorManager` methods of `DataProvider` return an `ExpressSpecifiedCursorManager`, which is an implementation of the `SpecifiedCursorManager` interface.

When your application no longer needs a `SpecifiedCursorManager`, it should close it to free resources in the application and in Oracle OLAP. To close the `SpecifiedCursorManager`, call its `close` method.

Updating the CursorManagerSpecification for a CursorManager

If your application is using OLAP 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 `CursorManagerSpecification` objects for the `Source` are no longer valid. You need to create new `CursorManagerSpecification` objects for the changed `Source`.

After creating a new `CursorManagerSpecification`, 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 `CursorManagerSpecification` with the new `CursorManagerSpecification`. You can then create a new `Cursor` from the `CursorManager`.

To determine whether the `CursorManagerSpecification` for a `CursorManager` needs updating, call the `isSpecificationUpdateNeeded` method of the `CursorManager`. You can also use a `CursorManagerUpdateListener` to listen for events generated by changes in a `Source`. For more information, see "CursorManagerUpdateListener Class" on page 9-14.

Other Classes

This topic describes `CursorInput`, `CursorManagerUpdateListener`, and `CursorManagerUpdateEvent` classes in the oracle.olapi.data.cursor package.

CursorInput Class

For Oracle OLAP in Oracle Database 10*g*, the OLAP API includes `Parameter` classes, which are more convenient than `CursorInput` objects. With a `Parameter`, you can create a parameterized `Source`. You can create a `CursorManagerSpecification` for a query that includes a parameterized `Source`, and then create a `CursorManager` and a `Cursor`.

You can then change the value of the `Parameter`, which changes the selection of dimension or measure elements specified by the parameterized `Source`. The `Cursor` for the query then has the new set of values for the changed query. You do not need to prepare and commit the `Transaction` again before getting the values of the `Cursor`. For information on parameterized `Source` objects, see Chapter 6, "Understanding Source Objects".

A CursorInput provides a value for a Source that you include in the array of Source objects that is the inputSources argument to the createCursorManager method of a DataProvider. If you create a CursorManagerSpecification for a Source that has one or more inputs, then you must provide an inputSources argument when you create a CursorManager for that CursorManagerSpecification. You include a Source in the inputSources array for each input of the Source that you pass to the createCursorManagerSpecification method.

When you create a CursorInput object, you can specify either a single value or a ValueCursor. If you specify a ValueCursor, then you can call the synchronize method of the CursorInput to make the value of the CursorInput be the current value of the ValueCursor.

CursorManagerUpdateListener Class

CursorManagerUpdateListener is an interface that has methods that receive CursorManagerUpdateEvent object. Oracle OLAP generates a CursorManagerUpdateEvent object in response to a change that occurs in a Source that is produced by a Template or when a CursorManager updates its CursorManagerSpecification. Your application can use a CursorManagerUpdateListener to listen for events that indicate it might need to create new Cursor objects from the CursorManager or to update its display of data from a Cursor.

To use a CursorManagerUpdateListener, implement the interface, create an instance of the class, and then add the CursorManagerUpdateListener to the CursorManager for a Source. When a change to the Source occurs, the CursorManager calls the appropriate method of the CursorManagerUpdateListener and passes it a CursorManagerUpdateEvent. Your application can then perform the tasks needed to generate new Cursor objects and update the display of values from the result set that the Source defines.

You can implement more than one version of the CursorManagerUpdateListener interface. You can add instances of them to the same CursorManager.

CursorManagerUpdateEvent Class

Oracle OLAP generates a CursorManagerUpdateEvent object in response to a change that occurs in a Source that is produced by a Template or when a CursorManager updates its CursorManagerSpecification.

You do not directly create instances of this class. Oracle OLAP generates `CursorManagerUpdateEvent` objects and passes them to the appropriate methods of any `CursorManagerUpdateListener` objects you have added to a `CursorManager`. The `CursorManagerUpdateEvent` has a field that indicates the type of event that occurred. A `CursorManagerUpdateEvent` has methods you can use to get information about it.

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 9-23.

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 9-5, 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
context.println("ValueCursor Position  Value ");
context.println("--------------------  ----------------------- ");
do
{
  context.println("          " + prodSelValCursor.getPosition() +
                  "           " + prodSelValCursor.getCurrentValue());
} while(prodSelValCursor.next());
```

The preceding example displays the following:

```
ValueCursor Position  Value
------------------  ------------------------
          1           PRODUCT_ROLLUP::ITEM::13
          2           PRODUCT_ROLLUP::ITEM::14
          3           PRODUCT_ROLLUP::ITEM::15
```

The following example sets the current position of `prodSelValCursor` to 2 and retrieves the value at that position.

```
prodSelValCursor.setPosition(2);
context.println(prodSelValCursor.getCurrentString());
```

The preceding example displays the following:

```
PRODUCT_ROLLUP::ITEM::14
```

For more examples of getting the current value of a `ValueCursor`, see Chapter 10.

Positions of a CompoundCursor

A `CompoundCursor` has one position for each set of the elements of its descendent `ValueCursor` objects. The current position of the `CompoundCursor` specifies one of those sets.

For example, `querySource`, the `Source` created in Example 9–1, 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.

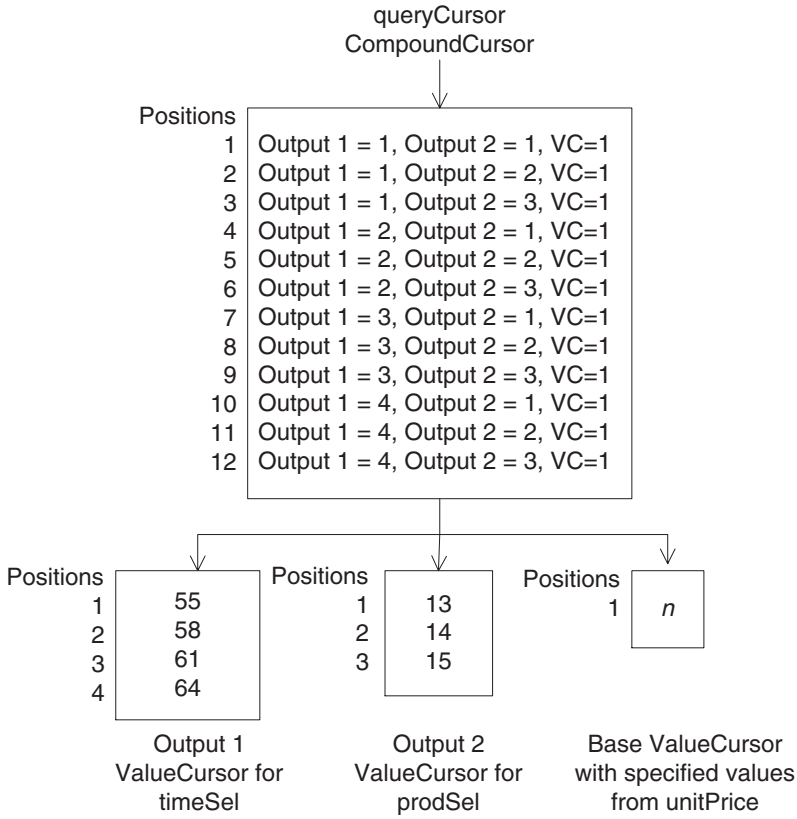
Each position of `queryCursor` specifies one set of positions of its outputs and its base `ValueCursor`. For example, position 1 of `queryCursor` defines the following set of positions for its outputs and its base `ValueCursor`:

- 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 9–2 illustrates the positions of `queryCursor` `CompoundCursor`, its base `ValueCursor`, and its outputs.

Figure 9–2 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 like `querySource`, the `ValueCursor` of its `Cursor` has only one value, and therefore only one position, at a time for any one position of the root `CompoundCursor`.

Figure 9–3 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 9–3 Crosstab Display of queryCursor

	Product		
Month	**13**	**14**	**15**
55	2426.07	3223.28	3042.22
58	2412.42	3107.65	3026.12
61	2505.57	3155.91	2892.18
64	2337.30	3105.53	2856.86

A `CompoundCursor` coordinates the positions of its `ValueCursor` objects relative to each other. The current position of the `CompoundCursor` specifies the current positions of its descendent `ValueCursor` objects. Example 9–2 sets the position of `queryCursor` and then gets the current values and the positions of the child `Cursor` objects.

Example 9–2 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);
System.out.println("CompoundCursor position set to " + pos + ".");
System.out.println("The current position of the CompoundCursor is = "
                   + rootCursor.getPosition() + ".");
System.out.println("Output 1 position = " + output1.getPosition() +
                   ", value = " + output1.getCurrentValue());
System.out.println("Output 2 position = " + output2.getPosition() +
                   ", value = " + output2.getCurrentValue());
System.out.println("VC position = " + baseValueCursor.getPosition() +
                   ", value = " + baseValueCursor.getCurrentValue());
```

Example 9–2 displays the following:

```
CompoundCursor position set to 5.
The current position of the CompoundCursor is 5.
Output 1 position = 2, value = CALENDAR::MONTH::58
Output 2 position = 2, value = PRODUCT_ROLLUP::ITEM::14
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 its slower varying output. For example, if the price of units for product 15 for month 64 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 9–3 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 9–1. 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 prepares and commits the current `Transaction`. The `TransactionProvider` in the example is `tp`. For information on `Transaction` objects, see Chapter 8.

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 9–3 Positions in an Asymmetric Query

```
// Create the query
querySource2 = prodSel.join(unitPrice).join(timeSel);

// Prepare and commit the current Transaction.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  output.println("Cannot commit current Transaction " + e);
}
tp.commitCurrentTransaction();

// Create the Cursor. The DataProvider is dp.
CursorManagerSpecification cursorMngrSpec =
                  dp.createCursorManagerSpecification(querySource2);
CursorManager cursorManager = dp.createCursorManager(cursorMngrSpec);
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.
System.out.println("CompoundCursor Output ValueCursor" +
                   "    ValueCursor");
System.out.println("  position      position | value  " +
                   "position | value");
do
{
System.out.println("      " + rootCursor.getPosition() +
                "             " + output1.getPosition() +
                "       " + getLocalValue(output1.getCurrentString()) +
                "       " + baseValueCursor.getPosition() +  "       "
                  + getLocalValue(baseValueCursor.getCurrentString()));
}
while(queryCursor2.next());
```

Example 9–3 displays the following:

```
CompoundCursor Output ValueCursor   ValueCursor
  position      position | value  position | value
      1             1       55       1       13
      2             1       55       2       14
      3             1       55       3       15
      4             2       58       1       15
      5             3       61       1       14
      6             3       61       2       15
      7             4       64       1       13
      8             4       64       2       14
```

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 55, all three products have sales greater than one, but for time value 58, only one of the products does. The other two time values, 61 and 64, have two products that meet the criteria. Therefore, the `ValueCursor` for the `CompoundCursor` has three positions for time 55, only one position for time 58, and two positions for times 61 and 64.

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 its 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 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 9-25.

From the data on the client computer, however, you cannot determine at what position of its 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`.

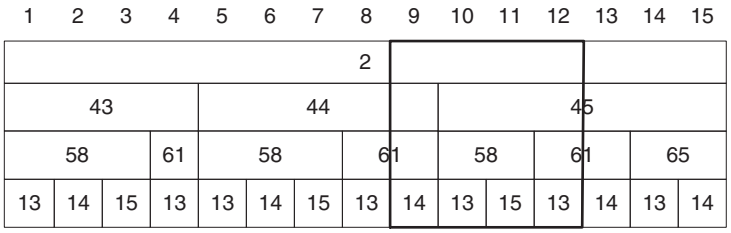
For example, suppose your application has a `Source` named `cube` that represents a cube that has an asymmetric edge. The cube has four outputs. The `cube Source` defines products with unit sales greater than one purchased by a certain customers during three months of the year 2001. The products were sold through the direct sales channel.

You create a `Cursor` for that `Source` and call it `cubeCursor`. The `CompoundCursor cubeCursor` has the following child `Cursor` objects:

- output 1, a `ValueCursor` for the channel values
- output 2, a `ValueCursor` for the time values
- output 4, a `ValueCursor` for the customer values
- The base `ValueCursor`, which has values that are the products with unit sales greater than one.

Figure 9–4 illustrates the parent, cubeCursor, with the values of its child Cursor objects layered horizontally. The slowest varying output, with the channel value, is at the top and the fastest varying child, with the product values, is at the bottom. The only portion of the edge that you are currently displaying in the user interface is the block between positions 9 and 12 of cubeCursor, which is shown within the bold border. The positions, 1 through 15, of cubeCursor appear over the top row.

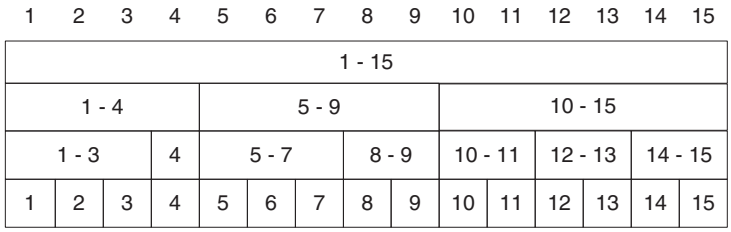
Figure 9–4 Values of the ValueCursor Children of cubeCursor



The current value of the output `ValueCursor` for the time `Source` is 44. You cannot determine from the data within the block that the starting and ending positions of the current value, 44, in the parent, `cubeCursor`, are 5 and 9, respectively.

The `cubeCursor` from the previous figure is shown again in Figure 9–5, this time with the range of the positions of the parent, `cubeCursor`, for each of the values of the child `Cursor` objects. By subtracting the smaller value from the larger value and adding one, you can compute the span of each value. For example, the span of the time value 44 is (9 - 5 + 1) = 5.

Figure 9–5 The Range of Positions of the Child Cursor Objects of cubeCursor



To specify that you want Oracle OLAP to calculate the starting and ending positions of a value of a child `Cursor` in its 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 10.

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. Figure 9–6 illustrates the number of positions of each child `Cursor` of `cubeCursor` relative to the value of its slower varying output. The child `Cursor` objects are layered horizontally with the slowest varying output at the top.

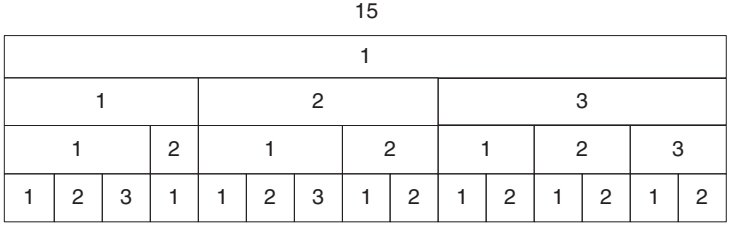
The total number of elements in `cubeCursor` is fifteen so the extent of `cubeCursor` is therefore fifteen. That number is over the top row of the figure. The

top row is the `ValueCursor` for the channel value. The extent of the `ValueCursor` for channel values is one because it has only one value.

The second row down is the `ValueCursor` for the time values. Its extent is 3, since there are 3 months values. The next row down is the `ValueCursor` for the customer values. The extent of its elements depends on the value of the slower varying output, which is time. The extent of the customers `ValueCursor` for the first month is two, for the second month it is two, and for the third month it is three.

The bottom row is the base `ValueCursor` for the `cubeCursor` `CompoundCursor`. Its values are products. The extent of the elements of the products `ValueCursor` depends on the values of the customers `ValueCursor` and the time `ValueCursor`. For example, since three products values are specified by the first set of month and customer values (products 13, 14, and 15 for customer 58 for time 43), the extent of the products `ValueCursor` for that set is 3. For the second set of values for customers and times (customer 61 for time 43), the extent of the products `ValueCursor` is 1, and so on.

Figure 9–6 The Number of Elements of the Child Cursor Objects of cubeCursor



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 10.

About Fetch Sizes

An OLAP 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 it. 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 6, "Understanding Source Objects". For examples of using parameterized Source objects, see Chapter 7, "Making Queries Using Source Methods".

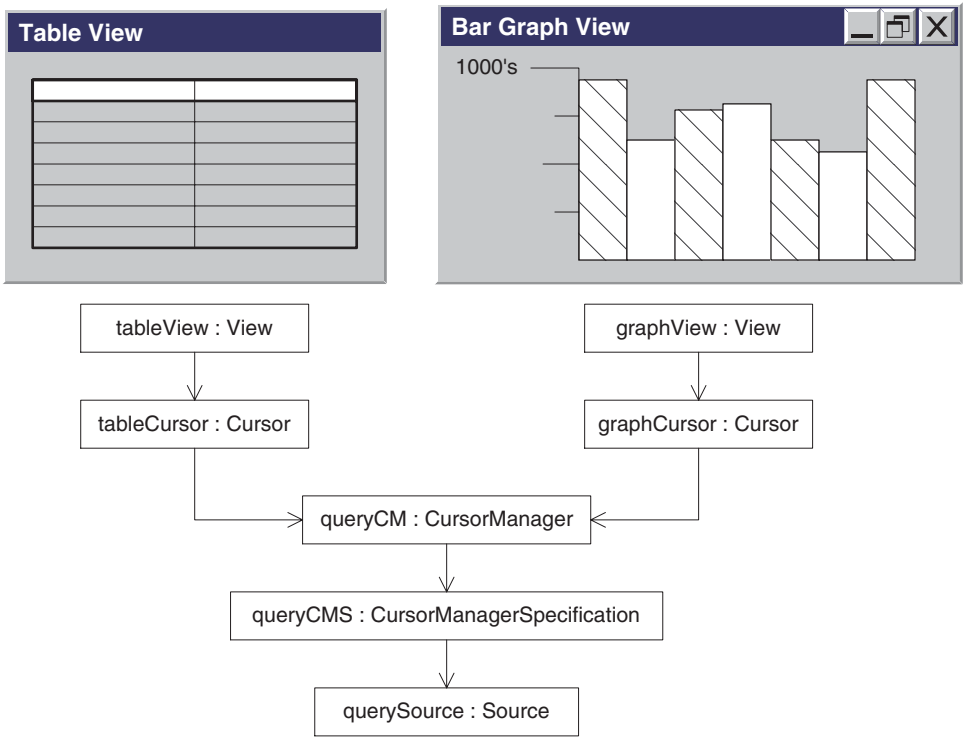
When you create a CursorManagerSpecification for a Source, as the first step in creating a Cursor, Oracle OLAP specifies a default fetch size on the root CursorSpecification of the CursorManagerSpecification. You can change the default fetch size with the setDefaultFetchSize method of the root CursorSpecification. You can also change the fetch size with the setFetchSize method of the CursorManager that you create using the CursorManagerSpecification, or with the setFetchSize method of a Cursor that you create with the CursorManager.

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 CursorManagerSpecification for a Source and then creates a CursorManager for the CursorManagerSpecification. 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 9–7 illustrates the relationship between the `Source`, the `Cursor` objects, and the views.

Figure 9–7 A Source and Two Cursors for Different Views of Its Values



10

Retrieving Query Results

This chapter describes how to retrieve the results of a query with an Oracle OLAP 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 9, "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

For the complete code of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

Retrieving the Results of a Query

A query is an OLAP API `Source` that specifies the data that you want to retrieve from Oracle OLAP and any calculations you want Oracle OLAP to perform on that 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 6, "Understanding Source Objects".

2. If the Source is a derived Source, prepare and commit the Transaction in which you created the Source. To prepare and commit the Transaction, call the prepareCurrentTransaction and commitCurrentTransaction methods of your TransactionProvider. For more information on preparing and committing a Transaction, see Chapter 8, "Using a TransactionProvider". If the Source is a primary Source, then you do not need to prepare and commit the Transaction.
3. Create a CursorManagerSpecification by calling the createCursorManagerSpecification method of your DataProvider and passing that method the Source.
4. Create a SpecifiedCursorManager by calling the createCursorManager method of your DataProvider and passing that method the CursorManagerSpecification. If the Source for the CursorManagerSpecification has one or more inputs, then you must also pass an array of Source objects that provides a Source for each input.
5. Create a Cursor by calling the createCursor method of the CursorManager. If you created the CursorManager with an array of input Source objects, then you must also pass an array of CursorInput objects that provides a value for each input Source.

Example 10–1 creates a Cursor for the derived Source named querySource. The example uses a TransactionProvider named tp and a DataProvider named dp. The example creates a CursorManagerSpecification named cursorMngrSpec, a SpecifiedCursorManager named cursorMngr, and a Cursor named queryCursor.

Finally, the example closes the SpecifiedCursorManager. When you have finished using the Cursor, you should close the SpecifiedCursorManager to free resources.

Example 10–1 Creating a Cursor

```
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  System.out.println("Cannot commit the current Transaction. " + e);
}
tp.commitCurrentTransaction();
```

```
CursorManagerSpecification cursorMngrSpec =
                 dp.createCursorManagerSpecification(querySource);
SpecifiedCursorManager cursorMngr =
                           dp.createCursorManager(cursorMngrSpec);
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 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 10–4. For an example of a `Source` that has nested output values, see Example 10–5.

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 10–2 gets the `Source` from `mdmProdHier`, which is an `MdmPrimaryDimension` 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 10–2 Getting a Single Value from a ValueCursor

```
Source prodSource = mdmProdHier.getSource();
// Because prodSource is a primary Source, you do not need to
// prepare and commit the current Transaction.
CursorManagerSpecification cursorMngrSpec =
               dp.createCursorManagerSpecification(productSource);
SpecifiedCursorManager cursorMngr =
                           dp.createCursorManager(cursorMngrSpec);
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 String value at the current
// position.
String value = prodValues.getCurrentString();

// Do something with the value, such as display it...

// Close the SpecifiedCursorManager.
cursorMngr.close();
```

Example 10–3 uses the same `Cursor` as Example 10–2. Example 10–3 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 10–3 Getting All of the Values from a ValueCursor

```
// prodValues is the ValueCursor for prodSource
prodValues.setPosition(1);
do
{
   System.out.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 10–4 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 10–4 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` and `tp` is the `TransactionProvider`.

Example 10–4 Getting ValueCursor Objects from a CompoundCursor

```
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
```

```
// Prepare and commit the current Transaction
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  System.out.println("Cannot commit the current Transaction. " + e );
}
tp.commitCurrentTransaction();

// Create a Cursor for unitsForSelections
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitsForSelections);
SpecifiedCursorManager cursorMngr =
                              dp.createCursorManager(cursorMngrSpec);
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
// SpecifiedCursorManager.
cursorMngr.close();
```

Example 10–5 uses the same units measure as Example 10–4, but it joins the dimension selections to the measure differently. Example 10–5 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 as its outputs has three child `ValueCursor` objects and one child `CompoundCursor`.

Example 10–5 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 its base values and channel values as the values of its 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 prepares and 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` and `tp` is the `TransactionProvider`.

Example 10–5 Getting Values from a CompoundCursor with Nested Outputs

```
// ...in someMethod...
Source custByChanSel = custSel.join(chanSel
Source unitsForSelections = units.join(prodSel)
                                  .join(timeSel)
                                  .join(custByChanSel);
// Prepare and commit the current Transaction
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  System.out.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for unitsForSelections
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitsForSelections);
SpecifiedCursorManager cursorMngr =
                              dp.createCursorManager(cursorMngrSpec);
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...
```

```
// 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
  {
     cpw.print(i++ + ": ");
     _printTuple(rootCursor);
     cpw.print("\n");
     cpw.flush();
  } while(rootCursor.next());
}

// If the Cursor passed to the _printTuple method is a ValueCursor,
// the method prints the value at the current position of the ValueCursor.
// If the Cursor passed in is a CompoundCursor, 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
    cpw.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
      cpw.print(",");
      _printTuple((Cursor)iterOutputs.next());
    }
    // Put a comma after the value of the last output
    cpw.print(",");
```

```
    // Get the base ValueCursor
    _printTuple(compoundCursor.getValueCursor());

    // Put a close parenthesis after the base value to indicate
    // the end of the tuple.
    cpw.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, its structure and get the values from its `ValueCursor` descendents. 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 10–6 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 prepares and 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 `TransactionProvider` is `tp` and the `DataProvider` is `dp`. The `cpw` object is a `PrintWriter`.

Example 10–6 Navigating for a Table View

```
Source unitPriceByMonth = unitPrice.join(productSel)
                                   .join(timeSel);
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  cpw.println("Cannot prepare the current Transaction. " + e);
}
tp.commitCurrentTransaction();

// Create a Cursor for unitPriceByMonth
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitPriceByMonth);
SpecifiedCursorManager cursorMngr =
                              dp.createCursorManager(cursorMngrSpec);
(CompoundCursor) rootCursor =
                         (CompoundCursor) cursorMngr.createCursor();

// Determine a starting position and the number of rows to display
int start = 7;
int numRows =12;

cpw.println("Month     Product     Unit Price");
cpw.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();
  cpw.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 10–6 displays the following table, which has the values specified by positions 7 through 18 of the `CompoundCursor`.

```
Month     Product     Unit Price
------     -------     ----------
  61         13        2505.57
  61         14        3155.91
  61         15        2892.18
  64         13        2337.30
  64         14        3105.53
  64         15        2856.86
  69         13        4281.42
  69         14        6017.90
  69         15        5793.54
  72         13        4261.76
  72         14        5907.92
  72         15        5760.78
```

Example 10–7 uses the same query as Example 10–6. 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 `TransactionProvider` is `tp` and the `DataProvider` is `dp`. The `cpw` object is a `PrintWriter`.

Example 10–7 Navigating for a Crosstab View without Pages

```
Source unitPriceByMonth = unitPrice.join(productSel)
                                   .join(timeSel);
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  cpw.println("Cannot prepare the current Transaction. " + e);
}
tp.commitCurrentTransaction();

// Create a Cursor for unitPriceByMonth
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitPriceByMonth);
SpecifiedCursorManager cursorMngr =
                         dp.createCursorManager(cursorMngrSpec);
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.
cpw.println("\t        Product");
cpw.println("\t-----------------------");
cpw.print("Month");
do
{
  String value = ((ValueCursor) columnCursor).getCurrentString();
  cpw.print("\t" + getLocalValue(value) + "  ");
} while (columnCursor.next());
cpw.println();
cpw.println("-----\t-------\t-------\t-------");

// Reset the column Cursor to its first element.
columnCursor.setPosition(1);
```

```
do
{
  // Print the row dimension values.
  String value = ((ValueCursor) rowCursor).getCurrentString();
  cpw.print(getLocalValue(value) + "\t");
  // Loop over columns
  do
  {
    // Print data value
    cpw.print(unitPriceValues.getCurrentValue() + "\t");
  } while (columnCursor.next());

  cpw.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  13       14       15
-----  -------  -------  -------
55     2426.07  3223.28  3042.22
58     2412.42  3107.65  3026.12
61     2505.57  3155.91  2892.18
64     2337.30  3105.53  2856.86
69     4281.42  6017.90  5793.54
72     4261.76  5907.92  5760.78
75     4149.12  6004.68  5730.28
78     3843.24  5887.92  5701.76
```

Example 10–8 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 its 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 13, 14, and 15). 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 58, 61, and 65). Those three values are the row headings. The page dimensions are selections of three time values (the months 43, 44, and 45), and one channel value (2, which is the direct sales channel).

The `TransactionProvider` is `tp` and the `DataProvider` is `dp`. The `cpw` object is a `PrintWriter`. The `getLocalValue` method gets the local value from a unique dimension value.

Example 10–8 Navigating for a Crosstab View with Pages

```
// ...in someMethod...
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  cpw.println("Cannot prepare the current Transaction. " + e);
}
tp.commitCurrentTransaction();

// Create a Cursor for unitsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(unitsForSelections);
SpecifiedCursorManager cursorMngr =
                              dp.createCursorManager(cursorMngrSpec);
```

```
CompoundCursor unitsForSelCursor = (CompoundCursor)
                                    cursorMngr.createCursor();

// Send the Cursor to the printAsCrosstab method
printAsCrosstab(unitsForSelCursor);

cursorMngr.close();
// ...the remainder of the code of someMethod...

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
      cpw.println();
      cpw.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();
    cpw.println(getLocalValue(value));
  }
  cpw.println();
}
```

```
// Prints the column headings on each page
private void printColumnHeadings(Cursor colCursor)
{
  do
  {
     cpw.print("\t");
     String value = ((ValueCursor) colCursor).getCurrentString();
     cpw.print(getLocalValue(value));
  } while (colCursor.next());
  cpw.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();
    cpw.print(getLocalValue(value));
    cpw.print("\t");
    // Loop over columns
    do
    {
      // Print data value
      cpw.print(dataCursor.getCurrentValue());
      cpw.print("\t");
    } while (colCursor.next());
    cpw.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 10–8 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   2
Customer 43

             Product
         ----------------
Month    13      14      15
-----    --      --      --
58        2       4       2
61        2       1       1
65        1       0       0


Channel   2
Customer 44

             Product
         ----------------
Month    13      14      15
-----    --      --      --
58        6       6       5
61        2       2       1
65        1       1       1


Channel   2
Customer 45

             Product
         ----------------
Month    13      14      15
-----    --      --      --
58        2       0       2
61        3       2       0
65        2       2       0
```

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 Oracle OLAP calculates the **extent** of the Cursor. The extent is the total number of positions of the Cursor. If the Cursor is a child Cursor of a CompoundCursor, its extent is relative to any slower varying outputs.
- Whether 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 the CursorSpecification for that Cursor. To get the CursorSpecification for a Cursor, you use methods of the CursorManagerSpecification that you create for a Source.

> **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, CursorSpecification, and CursorManagerSpecification objects or the concepts of fetch size, extent, or Cursor positions, see Chapter 9.

Example 10–9 creates a Source, creates a CursorManagerSpecification for the Source, and then gets the CursorSpecification objects from a CursorManagerSpecification. The root CursorSpecification is the CursorSpecification for the top-level CompoundCursor.

Example 10–9 Getting CursorSpecification Objects from a CursorManagerSpecification

```
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
```

```
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  System.out.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for unitsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(unitsForSelections);

// Get the root CursorSpecification of the CursorManagerSpecification.
CompoundCursorSpecification rootCursorSpec =
(CompoundCursorSpecification) cursorMngrSpec.getRootCursorSpecification();

// Get the CursorSpecification for the base values
ValueCursorSpecification baseValueSpec =
                        rootCursorSpec.getValueCursorSpecification();

// Get the CursorSpecification objects for the outputs
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification promoSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(0);
ValueCursorSpecification chanSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(1);
ValueCursorSpecification timeSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(2);
ValueCursorSpecification prodSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(3);
ValueCursorSpecification custSelValCSpec =
                       (ValueCursorSpecification) outputSpecs.get(4);
```

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 its child `Cursor` components. You might also want to know the position at which the current value of a child `Cursor` starts in its 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 its 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 its 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 10–10 specifies calculating the extent of a Cursor. The example uses the CursorManagerSpecification from Example 10–9.

Example 10–10 Specifying the Calculation of the Extent of a Cursor

```
CompoundCursorSpecification rootCursorSpec = (CompoundCursorSpecification)
                          cursorMngrSpec.getRootCursorSpecification();
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 10–11 specifies calculating the starting and ending positions of the current value of a child Cursor in its parent Cursor. The example uses the CursorManagerSpecification from Example 10–9.

Example 10–11 Specifying the Calculation of Starting and Ending Positions in a Parent

```
CompoundCursorSpecification rootCursorSpec = (CompoundCursorSpecification)
                            cursorMngrSpec.getRootCursorSpecification();

// 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.

```
boolean isSet;
Iterator iterOutputSpecs = rootCursorSpec.getOutputs().iterator();
ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)
                                          iterOutputSpecs.next();
while(iterOutputSpecs.hasNext())
{
  isSet = valCursorSpec.isParentStartCalculationSpecified();
  isSet = valCursorSpec.isParentEndCalculationSpecified();
  valCursorSpec = (ValueCursorSpecification) iterOutputSpecs.next();
}
```

Example 10–12 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 10–8.

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 `TransactionProvider` is `tp`, the `DataProvider` is `dp`, and `cpw` is a `PrintWriter`.

Example 10–12 Calculating the Span of the Positions in the Parent of a Value

```
Source unitsForSelections = units.join(prodSel)
                                 .join(custSel)
                                 .join(timeSel)
                                 .join(chanSel);
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  cpw.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();
```

```
// Create a CursorManagerSpecification for unitsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(unitsForSelections);

 // Get the root CursorSpecification from the CursorManagerSpecification.
CompoundCursorSpecification rootCursorSpec = (CompoundCursorSpecification)
                           cursorMngrSpec.getRootCursorSpecification();
// Get the CursorSpecification objects for the outputs
List outputSpecs = rootCursorSpec.getOutputs();
ValueCursorSpecification timeSelValCSpec =
  (ValueCursorSpecification) outputSpecs.get(3); // output for time
ValueCursorSpecification prodSelValCSpec =
 (ValueCursorSpecification) outputSpecs.get(1);  // 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
SpecifiedCursorManager cursorMngr = dp.createCursorManager(cursorMngrSpec);
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.
cpw.print(promoSelVals.getCurrentValue() + ", ");
cpw.print(chanSelVals.getCurrentValue() + ", ");
cpw.print(timeSelVals.getCurrentValue() + ", ");
cpw.print(custSelVals.getCurrentValue() + ", ");
cpw.print(prodSelVals.getCurrentValue() + ", ");
cpw.println(baseValCursor.getCurrentValue());
```

```
// Determine the span of the values of the two fastest varying outputs
int span;
span = (prodSelVals.getParentEnd() - prodSelVals.getParentStart()) +1);
cpw.println("The span of " + prodSelVals.getCurrentValue() +
            " at the current position is " + span + ".")
span = (timeSelVals.getParentEnd() - timeSelVals.getParentStart()) +1);
cpw.println("The span of " + timeSelVals.getCurrentValue() +
            " at the current position is " + span + ".")
cursorMngr.close();
```

This example displays the following text.

```
CHANNEL_ROLLUP::CHANNEL::2, CALENDAR::MONTH::44,
SHIPMENTS_ROLLUP::SHIP_TO::61, PRODUCT_ROLLUP::ITEM::15, 1.0
The span of PRODUCT_ROLLUP::ITEM::15 at the current position is 1.
The span of CALENDAR::MONTH::44 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`. You can set the fetch size on the root `Cursor` for a `Source`. `Cursor` for that `CursorSpecification` to change the fetch size of the `Cursor`. The default fetch size is 100.

Example 10–13 uses the `CursorManagerSpecification` from Example 10–9. It gets the default fetch size from the root `CursorSpecification`, creates a `Cursor` and sets a different fetch size on it, and then gets the fetch size for the `Cursor`. The `TransactionProvider` is `tp`, the `DataProvider` is `dp`, and `cpw` is a `PrintWriter`.

Example 10–13 Specifying a Fetch Size

```
CursorSpecification rootCursorSpec =
                        cursorMngrSpec.getRootCursorSpecification();
context.println("The default fetch size is "
               + rootCursorSpec.getDefaultFetchSize() + ".");
CursorManager cursorMngr = dp.createCursorManager(cursorMngrSpec);
Cursor rootCursor = cursorMngr.createCursor();
rootCursor.setFetchSize(10);
context.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.
```

11

Creating Dynamic Queries

This chapter describes the Oracle OLAP API `Template` class and its related classes, which you use to create dynamic queries. This chapter also provides examples of implementations of those classes.

This chapter includes the following topics:

- About Template Objects
- Overview of Template and Related Classes
- Designing and Implementing a Template

For the complete code of the examples in this chapter, see the example programs available from the Overview of the *Oracle OLAP Java API Reference*.

About Template Objects

The `Template` class is the basis of a very powerful feature of the Oracle OLAP 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 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 8, "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, a SourceDefinition is automatically created. The SourceDefinition has information about how the Source was created. Once created, the Source and its SourceDefinition are paired immutably. The getSource method of a SourceDefinition returns its paired Source.

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 paired 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 different than 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 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 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 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 other classes that Oracle OLAP creates, work together to produce the Source that the Template defines. The classes that Oracle OLAP provides, which you create by calling factory methods, are the following:

- DataProvider
- DynamicDefinition

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 immutably paired SourceDefinition of that 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 11–1 on page 11-7.

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 11–2 on page 11-10.

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 11–3 on page 11-11.

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 paired to its 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 paired to the Source that the generateSource method currently returns. For an example of the use of a DynamicDefinition, see Example 11–4 on page 11-13.

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 Single Selections 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. From the Single Selections dialog box, the end user 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 11–1, Example 11–2, Example 11–3, and Example 11–4. The `TopBottomTemplateState` and `TopBottomTemplateGenerator` classes are implemented as inner classes of the `TopBottomTemplate` outer class.

Implementing the Classes for a Template

Example 11–1 is an implementation of the `TopBottomTemplate` class.

Example 11–1 Implementing a Template

```
package globalExamples;

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 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 _definition;

  /**
   * 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.
    _definition =
    createDynamicDefinition(new TopBottomTemplateGenerator(dataProvider));
  }

  /**
   * Gets the Source produced by the TopBottomTemplateGenerator
   * from the DynamicDefinition.
   */
  public final Source getSource()
  {
    return _definition.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 11–2 is an implementation of the `TopBottomTemplateState` inner class.

Example 11–2 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 11–3 is an implementation of the `TopBottomTemplateGenerator` inner class.

Example 11–3 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 `Source` created by the `Template`. This section provides an example of an application that uses the `TopBottomTemplate` described in Example 11–1. For brevity, the code does not contain much exception handling.

The `Context10g` class used in the example has methods that do the following:

- Connects to an Oracle Database instance as the specified user.
- Provides the OLAP Catalog metadata objects for the measure and the dimensions selected by the end user.
- Creates `Cursor` objects and displays their values.

Example 11–4 does the following:

- Creates a `Context10g` object and from it gets the `DataProvider` and the `TransactionProvider`.
- From the `Context10g` object, gets the `MdmMeasure` and the `MdmPrimaryDimension` objects that it uses.
- Creates a `SingleSelectionTemplate` for selecting single values from some of the dimensions of the measure.
- Creates a `TopBottomTemplate` and stores selections made by the end user.
- Gets the `Source` produced by the `TopBottomTemplate`.
- Uses the `Context10g` object to create a `Cursor` for that `Source` and to display its values.

To use Example 8–2 from Chapter 8, replace the lines in the `run` method beginning with the following comment to the end of the method.

```
// Replace from here for the Using Child Transaction example.
```

Example 11–4 Getting the Source Produced by the Template

```
package globalExamples;

import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.StringSource;
import oracle.olapi.transaction.Transaction;
import oracle.olapi.transaction.TransactionProvider;
import oracle.olapi.transaction.NotCommittableException;

/**
 * Creates a query that specifies a number of values from the top or
 * bottom of a list of values from one of the dimensions of a measure.
 * The list is determined by the measure and by single values from
 * the other dimensions of the measure.
 * Displays the results of the query.
 */
public class TopBottomTest
{
  /**
   * Creates a Context object that connects to an Oracle Database instance.
   * Gets the MdmMeasure for the Global schema UNITS measure
   * and the MdmPrimaryDimension objects for that measure.
```

```
 * Gets the default hierarchies for the dimensions and then gets Source
 * objects for the specified levels of the hierarchies.
 * 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 results.
 */
public void run(String[] args)
{
  // Create a Context object and get the DataProvider from it.
  Context10g context = new Context10g(args, true);
  DataProvider dp = context.getDataProvider();
  TransactionProvider tp = context.getTransactionProvider();

  // Get the MdmMeasure for the measure.
  MdmMeasure mdmUnits = context.getMdmMeasureByName("UNITS");
  // Get the Source for the measure.
  Source units = mdmUnits.getSource();

  // Get the MdmPrimaryDimension objects for the dimensions of the measure.
  MdmPrimaryDimension[] mdmPrimDims =
    context.getMdmPrimaryDimensionsByName(new String[]
                                          {"CUSTOMER",
                                            "PRODUCT",
                                            "CHANNEL",
                                            "TIME"});
  MdmPrimaryDimension mdmCustDim = mdmPrimDims[0];
  MdmPrimaryDimension mdmProdDim = mdmPrimDims[1];
  MdmPrimaryDimension mdmChanDim = mdmPrimDims[2];
  MdmPrimaryDimension mdmTimeDim = mdmPrimDims[3];

  MdmHierarchy mdmShipRollup = mdmCustDim.getDefaultHierarchy();
  MdmHierarchy mdmProdRollup = mdmProdDim.getDefaultHierarchy();
  MdmHierarchy mdmChanRollup = mdmChanDim.getDefaultHierarchy();
  MdmHierarchy mdmTimeCal = mdmTimeDim.getDefaultHierarchy();

  StringSource shipRollup = (StringSource) mdmShipRollup.getSource();
  StringSource prodRollup = (StringSource) mdmProdRollup.getSource();
  StringSource chanRollup = (StringSource) mdmChanRollup.getSource();
  StringSource calendar = (StringSource) mdmTimeCal.getSource();

  // Create a SingleSelectionTemplate to produce a Source that
  // specifies a single value for each of the levels other
  // than the base dimension for the selected measure.
```

```
SingleSelectionTemplate singleSelections =
                    new SingleSelectionTemplate(units, dp);\

// Specify a unique value for each of the hierarchies.
// Region 8 is Asia Pacific.
singleSelections.addSelection(shipRollup, "SHIPMENTS_ROLLUP::REGION::8");
// Year 4 is 2001.
singleSelections.addSelection(calendar, "CALENDAR::YEAR::4");
// Channel 2 is Direct Sales.
singleSelections.addSelection(chanRollup,
                              "CHANNEL_ROLLUP::CHANNEL::2");

// Create a TopBottomTemplate specifying, as the base, the Source for a
// level of the default hierarchy of a dimension.
TopBottomTemplate topNBottom = new TopBottomTemplate(prodRollup, dp);

// Specify whether to retrieve the elements from the beginning (top) or
// the end (bottom) of the 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());

// Prepare and commit the current transaction.
try
{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e)
{
  context.println("Cannot prepare current Transaction. " + e);
}
tp.commitCurrentTransaction();

// Replace from here for the Using Child Transaction Objects example.
// Get the Source produced by the TopBottomTemplate,
// create a Cursor for it, and display the results.
context.println("The top ten products are:\n");
context.displayTopBottomResult(topNBottom.getSource());
```

```
  // Change the number of elements selected and the type of selection.
  topNBottom.setN(5);
  topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_BOTTOM);

  // Prepare and commit the current transaction.
  try
  {
    tp.prepareCurrentTransaction();
  }
  catch(NotCommittableException e)
  {
    context.println("Cannot prepare the current Transaction. " + e);
  }

  tp.commitCurrentTransaction();

  // Get the Source produced by the TopBottomTemplate,
  // create a Cursor for it, and display the results.
  context.println("\nThe bottom five products are:\n");
  context.displayTopBottomResult(topNBottom.getSource());
}

/**
 * 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 Context10g class.
 */
public static void main(String[] args)
{
  new TopBottomTest().run(args);
}
```

The `TopBottomTest` program produces the following output.

```
The top ten products are:

 1. PRODUCT_ROLLUP::TOTAL_PRODUCT::1
 2. PRODUCT_ROLLUP::CLASS::3
 3. PRODUCT_ROLLUP::FAMILY::7
 4. PRODUCT_ROLLUP::CLASS::2
 5. PRODUCT_ROLLUP::FAMILY::9
 6. PRODUCT_ROLLUP::FAMILY::6
 7. PRODUCT_ROLLUP::FAMILY::11
 8. PRODUCT_ROLLUP::ITEM::30
 9. PRODUCT_ROLLUP::ITEM::28
10. PRODUCT_ROLLUP::ITEM::47

The bottom five products are:

1. PRODUCT_ROLLUP::ITEM::36
2. PRODUCT_ROLLUP::ITEM::43
3. PRODUCT_ROLLUP::ITEM::44
4. PRODUCT_ROLLUP::ITEM::38
5. PRODUCT_ROLLUP::ITEM::22
```

A

Setting Up the Development Environment

This appendix describes the development environment for creating applications that use the OLAP API.

This appendix includes the following topics:

Overview

The Oracle Database installation, with the OLAP option, provides the OLAP API and other class libraries, as `jar` files, that you require to develop an OLAP 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 API class libraries.
- Certain Oracle JDBC (Java Database Connectivity) `jar` files, which provide communications between the application and the Oracle database. The Oracle installation includes the JDBC files. You must use these JDBC files and not those from another Oracle product or those from a product from another vender.

- The Java Development Kit (JDK) version 1.2. 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, 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 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/olapi/lib`
`classes12.jar`	`/jdbc/lib`
`nls_charset12.jar`	`/jdbc/lib`

The `olap_api.jar` file contain the OLAP API classes. The `classes12.jar` file contains JDBC classes for use with JDK 1.2.x. The `nls_charset12.jar` file contains JDBC classes that provide globalization support.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W