Oracle9*i* OLAP Services

Developer’s Guide to the Oracle OLAP API

Release 1 (9.0.1)

June 2001

Part No. A88756-01

Oracle9*i* OLAP Services Developer’s Guide to the Oracle OLAP API, Release 1 (9.0.1)

Part No. A88756-01

Contents

Send Us Your Comments

Oracle9*i* OLAP Services Developer's Guide to the Oracle OLAP API, Release 1 (9.0.1)

Part No. A88756-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. 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?

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

- FAX - 781-684-5880. Attn: Oracle9i OLAP Services
- Postal service:
  Oracle Corporation
  Oracle OLAP Services Documentation
  200 Fifth Avenue
  Waltham, MA 02451-8720
  USA

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

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

Preface

What this manual is about

The *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API* introduces Java programmers to the Oracle®OLAP API the Java application programming interface for Oracle OLAP Services. Through OLAP Services, the OLAP API provides access to data stored in an Oracle database. The OLAP API's capabilities for querying, manipulating, and presenting data are particularly suited to applications that perform Online Analytical Processing.

Intended audience

This manual is intended for Java programmers who are responsible for creating applications that perform Online Analytical Processing. It assumes that you are already familiar with Java, relational database management systems, data warehousing, and Online Analytical Processing (OLAP) concepts.

Before you begin

Before you can use the OLAP API you must set up the OLAP API client files in your Java development environment. You must also have access to an Oracle database instance. In addition, that database must include data that has been prepared as a data wardhouse and supplied with metadata using the OLAP management feature in Oracle Enterprise Manager.

Related information

For information on setting up the OLAP API client files, see Appendix A of this document. For information on installing Oracle with OLAP Services, see *Oracle 9i Installation Guide*. For information on data warehouse and metadata requirements, see *Oracle9i OLAP Services Concepts and Administration Guide*. For information on how to define the metadata used by OLAP Services, see the Oracle Enterprise Manager Help topics for the OLAP management feature.

Structure of this document

The *Oracle9i OLAP Services Developer's Guide to the Oracle OLAP API* is structured as follows:

- Chapter 1 provides an overview of Oracle OLAP Services and introduces the OLAP API.
- Chapter 2 describes the OLAP API metadata.
- Chapter 3 and Chapter 4 describe how to connect and discover the available metadata.
- Chapter 5 describes how to create specifications for queries.
- Chapter 6 describes how to select data.
- Chapter 7 describes how to perform calculations.
- Chapter 8 describes how to use OLAP API transactions.
- Chapter 9 and Chapter 10 describe how to use cursors to retrieve data into your application.
- Chapter 11 describes how to create and use templates to create dynamic queries.
- Appendix A provides information on how to set up the OLAP API client files.
- Appendix B provides information on how to use the Smart Agent Naming Service when making connections.

Related documentation

You will find the following documentation helpful when using the OLAP API and OLAP Services:

- *Oracle9i OLAP Services Concepts and Administration Guide* — Describes how to use OLAP Services. It introduces the basic concepts underlying business analysis and multidimensional querying, as well as the basic tools used for application development and system administration.
- *Oracle9i OLAP Services OLAP API Reference* — Provides online reference documentation for the OLAP API, the Java application programming interface for Oracle OLAP Services.

- *Oracle9i OLAP Services Developer's Guide to the OLAP DML* — Explains how application developers can perform complex data analysis tasks (such as forecasts, models, allocations, and some types of non-additive aggregation) by using the OLAP DML.
- *Oracle9i Data Warehousing Guide* — Discusses the database structures, concepts, and issues involved in creating a data warehouse to support OLAP solutions.

Conventions

Text conventions

You will find the following text conventions in this document.

Convention	Usage
Boldface text	Used for notes and other secondary information in tables (for example, **Note**).
`Fixed-width` text	Indicates Java package, interface, class, method, field (constant), and exception names. Also indicates examples and anything that you must type exactly as it appears.
Italic text	Indicates emphasis, a new term, and titles of documents.

Conventions for examples

The examples included in this document are simplified for the purpose of clarification. They do not include the error handling that is required for good programming style.

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. 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/`

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.

1 Introduction to the OLAP API

Chapter summary

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

List of topics

This chapter includes the following topics:

- OLAP API Overview
- Access to Data and Metadata Through the OLAP API
- OLAP API Software Components
- Developing an OLAP API Application
- Tasks That an OLAP API Application Performs

OLAP API Overview

What is the OLAP API?

The OLAP API is a Java application programming interface (API) through which an application can access data for online analytical processing (OLAP). It is the API that is supplied with OLAP Services, an Oracle 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 recreating 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 customer, product, and time. 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 Time 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 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 hierarchy. For example, a time 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, which is referred to as the query’s result set. The specification may require selecting, aggregating, calculating, or otherwise manipulating data. If such manipulation is required, it is an intrinsic part 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, sales data can be organized into a cube whose edges contain values from the time, product, and customer dimensions and whose body contains values from the sales 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 OLAP Services, 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 or snowflake schema, which represents a multidimensional data model. The star or snowflake 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 a star or snowflake schema. Then the database administrator must use the OLAP management feature in Oracle Enterprise Manager to create the required metadata, which can be defined as “data about the data.” Once the metadata is in place, an application can access both the data and the metadata through the OLAP API.

The collection of warehouse data for which a database administrator has created metadata using the OLAP management feature of Oracle Enterprise Manager is referred to as the data store to which the OLAP API gives access.

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 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 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.

As the programming interface for OLAP Services, the OLAP API is part of Oracle. The OLAP API development environment includes an Oracle database, an OLAP service, and the OLAP management feature in Oracle Enterprise Manager. The OLAP management feature in Oracle Enterprise Manager is an important companion to the OLAP API, because it generates the metadata that the OLAP API requires for accessing data that is stored in an Oracle database.

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.

Access to Data and Metadata Through the OLAP API

Distinction between data and metadata

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

- The fact that a given set of data exists. For example, a sales measure exists in the data store.

- The structure of that set of data. For example, the sales measure is dimensioned by customer, product, and time.
- The characteristics of that set of data. For example, the sales measure contains numeric values, and it has a descriptive name that can be used in reports.

In contrast, the fact that 3542 dollars worth of boys outerwear was sold in Atlanta during January 1999 is data, not metadata.

These examples distinguish between the metadata and the data for a measure called Sales. 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 one of its elements is “boys outerwear” is data.

MDM model in the OLAP API

The OLAP API’s 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:

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

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 in `MdmMetadataProvider`. This method returns the top-level MdmSchema, which contains all the MdmMeasure and MdmDimension objects that are accessible through this particular MdmMetadataProvider. The MdmDimension and MdmMeasure objects might be organized in a hierarchical tree, with subschemas nested under the top-level schema. Using the getMeasures, getDimensions, and getSubSchemas methods on all 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 MdmHierarchy, MdmLevel, and MdmAttribute objects.

Chapter 2 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 called salesAmount might represent a set of numeric elements whose values are dollar sales figures, and an MdmDimension called productDim might represent a set of text elements whose values are product names. 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 query on that data. And an application must create a query in order to select, calculate, and otherwise manipulate data for analysis.

In order to create a query on the data for an MdmMeasure or MdmDimension, an application calls the getSource method on the MdmMeasure or MdmDimension. This method creates a Source object that represents the data for the purpose of querying. A Source is a specification for a query that 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 (January, February, March) or a calculation of the values of one MdmMeasure minus those of another (salesAmount minus unitCost). An application can use the powerful methods on Source and its subclasses to combine data in any way that the user requires. And each new query is a new Source.

When an application prepares to display the data for a given Source, it creates a Cursor for the Source. The application then uses this Cursor to request and retrieve the data from the OLAP service. 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). The OLAP service 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`
- `NumberSource`
- `StringSource`

One of the useful characteristice of `Source` objects is that they make no distinction between dimensions and measures. All `Source` objects behave in the same way.

Access to data in a relational data store

To be accessed through the OLAP API, relational data must be stored in an instance of an Oracle database that has OLAP Services installed with it. The data must be in a data warehouse, and it must be organized according to a star or snowflake schema. For information about creating a data warehouse and about star and snowflake schemas, see the *Oracle9i DataWarehousing Guide*.

A database administrator uses the OLAP management feature in Oracle Enterprise Manager to add metadata to the data warehouse. The OLAP management feature makes it possible to organize data within one or more folders, which are containers for measures that are related by subject matter. Sometimes measure folders are referred to as business areas. The data store for an OLAP service includes all of the measure folders that are defined in the OLAP management feature of Oracle Enterprise Manager.

Access to data in an analytic workspace

The OLAP API also provides the ability to work with data and metadata that is managed by OLAP Services in an analytic workspace. The procedures for using the OLAP API are essentially the same in either case. However, the preparation of the data and metadata is different.

For more information about using an analytic workspace, see the *Oracle9i OLAP Services Developer's Guide to the OLAP DML*.

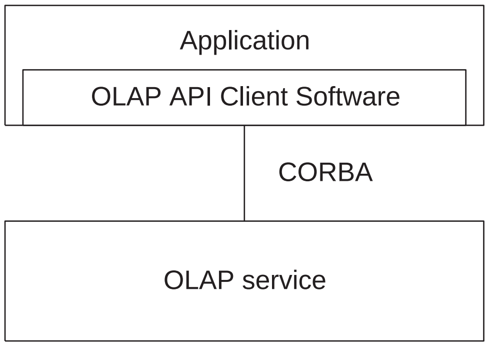
User connection requirements

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 OLAP Services and that users have database privileges that give them access to the data. For information about setting up an OLAP service for such connections, see the *Oracle9i OLAP Services Concepts and Administration Guide*.

OLAP API Software Components

Component overview

The OLAP API is the programming interface for OLAP Services. When a Java application calls methods on OLAP API Java classes, it uses the OLAP API client software to communicate with an OLAP service, which typically resides on a different computer. An OLAP service is a child process of an Oracle database instance. The communication between the OLAP API client software and an OLAP service is provided through Common Object Request Broker Architecture (CORBA). The following diagram shows the application using the OLAP API client software to communicate with the OLAP service.



An application that uses the OLAP API client software (that is, calls methods in OLAP API classes) can reside on a single computer, or it can be divided into separate parts on different computers. For example, the end-user portion can be separate from the portion that makes OLAP API calls. In this case, software on three computers could be involved.

OLAP API client software

The OLAP API client software is a set of Java packages containing classes that implement the programming interface to an OLAP service. An application calls the methods on these classes for discovering, querying, processing, and retrieving data.

To use the OLAP API classes as you develop your application, import them into your Java code in the standard way. When you deliver your application to users, include the OLAP API classes with the application. You must also deliver the CORBA implementation that is supplied with OLAP Services.

In order to develop an OLAP API application, you must have the Java Development Kit (JDK) 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. For detailed information about the OLAP API classes and methods, see the *Oracle9i OLAP Services OLAP API Reference* and subsequent chapters of this guide.

OLAP service software

The software that implements the server side of the OLAP API is an integral part of an OLAP service. It includes modules that interact with the OLAP API client software through CORBA, as well as software that interacts with the associated Oracle database. In response to requests from the OLAP API client software, the OLAP service does the following:

- Retrieves data from the database as needed and in the most efficient way possible
- Processes the data (for example, aggregating or calculating new values)
- Sends the data to the OLAP API client software in the most efficient block sizes possible

Because the server-side OLAP API software is an integral part of an OLAP service, no special installation steps are required to make it available. Furthermore, a CORBA implementation is provided with OLAP Services, in order to facilitate connections between the OLAP API client software and an OLAP service.

Before an application can access a data store through the OLAP API, the CORBA software must be running on the OLAP Services computer, and the connection between the OLAP service and the Oracle database must be in place.

For information about configuring an OLAP service for connecting with the database, see the *Oracle9i OLAP Services Concepts and Administration Guide*.

CORBA support

The OLAP API client software and an OLAP service communicate through an implementation of the CORBA specification. Using CORBA makes it possible for the Java-language client software to communicate with the OLAP service code, which is written in C++. It also maintains the object-oriented context in which the OLAP API was designed.

The CORBA implementation that is used by OLAP Services is VisiBroker from Borland. On the OLAP Services computer, VisiBroker for C++ is automatically installed. You, as an application developer, install the VisiBroker for Java files on your application development computer when you install the OLAP API as described in Appendix A. On most platforms, you also use the CORBA naming service that is built into Oracle for finding the OLAP service to which you will connect.

CORBA requires an Object Request Broker (ORB) both on the OLAP API client computer and on the OLAP Services computer. When an application calls a method that requires an interaction with an OLAP service, the client ORB intercepts the call, interacts with the OLAP Services ORB to find the object on the server side that can implement the request, passes the parameters, invokes the object's method, and returns the results.

Both the OLAP API client software and the OLAP service interact with their respective ORBs using the CORBA Interface Definition Language (IDL), which solves the problem of translation between Java and C++. The transport protocol that is used by a CORBA connection is Internet Inter-ORB Protocol (IIOP), which transfers messages between the ORBs over a TCP/IP connection.

For more information about the CORBA specification, explore the Object Management Group (OMG) Web pages at `www.omg.org`.

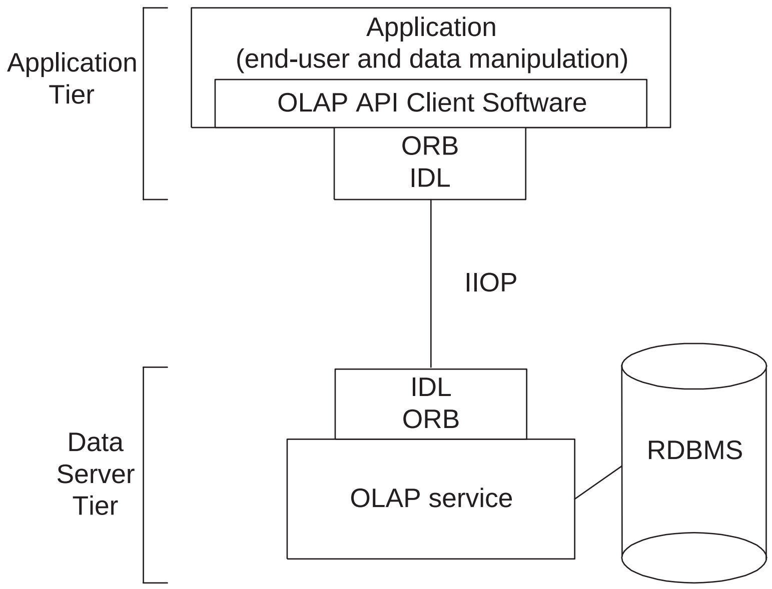
Application configurations

A typical OLAP API application is designed for either a two-tier or a three-tier configuration. Each tier represents an area of functionality, not a physical computer. That is, each tier can be implemented on a single computer or on more than one.

Two-tier configuration

In a two-tier configuration, the application is typically a standalone that provides the graphical user interface to the end user, as well as the OLAP API data manipulation capabilities. Thus, the two tiers for the configuration are the application tier and the data server tier. Communication between the two is through the OLAP API, using IIOP, as described in "CORBA support" on page 1-10.

The following diagram shows the two tiers and the IIOP communications between them.



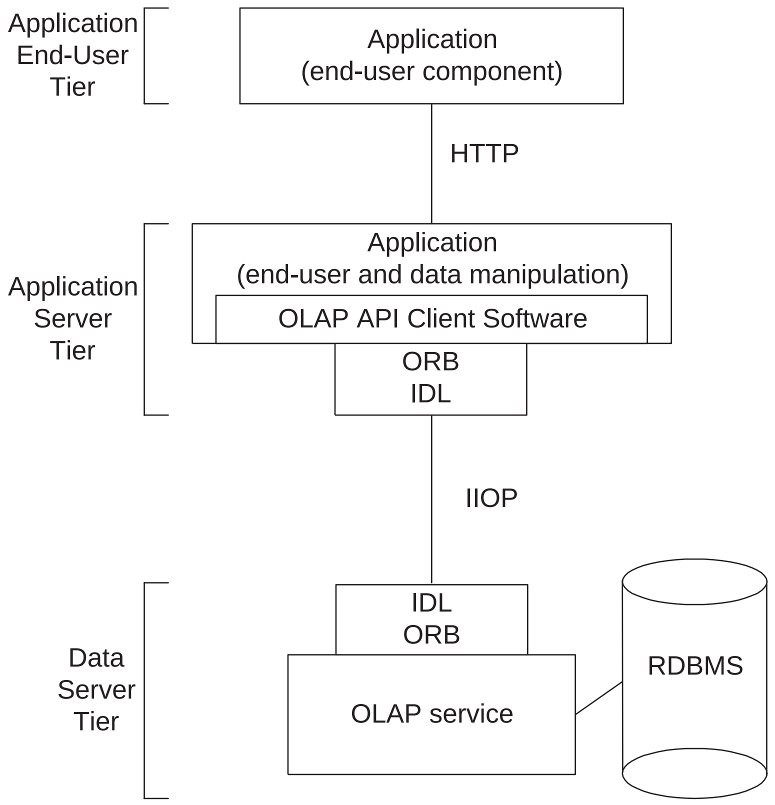
Three-tier configuration

In a three-tier configuration, the application is divided into two components. One provides the graphical user interface to the end user, and the other provides the OLAP API data manipulation capabilities. These two components represent the application end-user tier and the application server tier.

An application server is a functional component of an application. It performs a service, such as data manipulation, for multiple instances of the end-user component. For example, an application server might be called by a Web server to respond to requests from end users who are accessing Web pages.

In addition to the two application-specific tiers, the three-tier configuration includes the data server tier with the OLAP service and the Oracle database. Communication between the application server tier and the data server tier is through the OLAP API using IIOP, as described in "CORBA support" on page 1-10. Communication between the two application tiers is completely up to you, as the application developer. For example, a Web application might use Hypertext Transfer Protocol (HTTP), and it might employ a Web server to make the connection.

The following diagram shows the three tiers and the communications among them.


Application End-User Tier
Application (end-user component)
HTTP
Application Server Tier
Application (end-user and data manipulation)
OLAP API Client Software
ORB
IDL
IIOP
Data Server Tier
IDL
ORB
OLAP service
RDBMS

Developing an OLAP API Application

Overview of the development process

As an application developer, you perform the following steps to create an OLAP API application:

1. Decide on general design issues.
2. Decide on requirements for end-user queries.
3. Design OLAP API `Template` objects that create end-user queries. This is an optional step.
4. Write and test the Java code for the application.
5. Deploy the application to users.

The rest of this topic presents a general description of each step.

Step 1: Decide on general design issues

Consider broad questions such as the following:

- Will the application be a standalone application (two-tier architecture), or will it be divided, with end-user code on a separate tier from the data manipulation code (three-tier architecture)?
- Will the application always access the same known metadata (for example, describing employee data whose structure is constant), or must it discover what metadata is available every time it makes a connection?

Step 2: Decide on requirements for end-user queries

Specify, in as much detail as possible, the nature of the queries that the end user will be able to make. Because the OLAP API makes it possible to define queries in a step-by-step process, it is also important to decide on the query modification capabilities that the application will offer the user. Consider questions such as the following:

- By what criteria will the end user select data through the application's dialog boxes? For example, will the application present a list of dimensions? Can the user drill up and down on the hierarchy of a dimension? Are there attributes of dimensions that the user can specify for selecting data (for example, color or size)? Can the user make selections based on data values (for example, population over 20,000)?

- As the user refines a query through a series of steps, can the user undo a step in the process to return the query to an earlier state?
- As the user refines a query, can the user specify the scope of an undo request? For example, the undo request might apply only to the values of one field out of many in the selection dialog box.

Planning the end-user queries is a crucial step in the application design process, so you should complete it as thoroughly as possible. Ideally, you should create an end-user query model that identifies all the conceptual query objects with which the application user interface will deal. This strategy takes advantage of the strengths of object-oriented design, and it allows for a clear correspondence between user interface objects and OLAP API objects.

The following are examples of conceptual query objects for an application user interface:

- Dimension. This object has hierarchies on which the user can drill and attributes from which the user can select.
- Dimension selection. This object represents a selection of dimension elements.
- Edge. This object represents one side of a cube and has related dimension objects.
- Cube. This multidimensional object has related edge objects. It also has a related measure.

Each of these conceptual query objects can be represented by an OLAP API `Template` object.

Step 3: Design OLAP API Template objects that create end-user queries

An optional step in implementing an OLAP API application is designing `Template` objects. This step is recommended because, the use of `Template` objects offers the following benefits:

- Dynamic queries. With a `Template`, you can create a modifiable query. That is, when you have created one query and you want to execute another one that is similar but not identical, you do not have to create an entirely new query. You simply make a small change to the existing query. Thus, the query is dynamic, rather than static.

- Refinement and rollback of queries. With a `Template`, you can capture a series of steps that a user has completed when specifying a query. Each step refines the query further and is recorded as a new query state. If the user decides to cancel one or more of the specification steps, you can rollback the query to an earlier state.
- Matching of code to user interface characteristics. When you design a `Template`, you can make it correspond directly to the operations that a user performs. For example, if your application includes a balance sheet, you can create a balance sheet `Template` that incorporates all the appropriate characteristics (such as a method of aggregation) and behaviors (such as automatic totalling).

For a more detailed example of how `Template` objects mirror the query-building aspects of an application's user interface, imagine an application that allows the user to create a three-dimensioned cube of data through the following steps:

1. Choose a measure whose data will be in the cube.
2. Select the values for each dimension that will provide structure to the cube.
3. Specify the placement of the dimensions on the three edges of the cube.

As the application developer for this interface, you would design a `Template` subclass for each of the following objects: dimension, dimension selection, edge, and cube. As part of the design, you would specify methods on the `Template` subclasses that allow you to combine objects as needed. For example, the edge `Template` class might have an `addDimension` method, and the cube `Template` class might have an `addEdge` method. Once you have implemented the dimension, dimension selection, edge, and cube `Template` classes, you can use them again and again in your application. They are basic building blocks in your application's code for querying and manipulating data.

In this stage of the application design process, you should make detailed specifications for each `Template` in the application. For information about designing `Template` objects, see Chapter 11.

Step 4: Write and test the Java code for the application

Up to this step, you have not written any Java code. You have considered questions about the design of your application, and you have made detailed specifications for the `Template` objects that your application will include. Now you must do the following to implement the application:

1. Set up the OLAP API client software on your development computer, as described in Appendix A. If you are designing a three-tiered application, the development computer (from the OLAP API point of view) is the middle-tier computer.
2. Identify the data store that you will use for developing and testing the application. Ensure that the data is structured as a star or snowflake schema in an Oracle data warehouse, and ensure that the OLAP management feature in Oracle Enterprise Manager has provided the metadata.
3. Write the Java classes for your application, importing the OLAP API classes as needed. Among the Java classes that you write, include the `Template` classes that you designed.
4. Test your application using the test data store.

For information about coding an application that uses the OLAP API, see the subsequent chapters of this guide and the *Oracle9i OLAP Services OLAP API Reference*. See"Tasks That an OLAP API Application Performs" on page 1-17 for a description of the tasks that an application typically performs.

Step 5: Deploy the application to users

Keep the following in mind when you deploy your application:

- Include the OLAP API Java classes along with the ones that you have developed.
- Ensure that the user's computer (or the middle tier computer) has access to an OLAP service using CORBA.
- Ensure that the user has access to an appropriate Oracle data warehouse with metadata prepared by the OLAP management feature in Oracle Enterprise Manager. Access must be through the OLAP service.
- Provide documentation for your application, giving installation instructions and explaining the user interface that you have created.

Tasks That an OLAP API Application Performs

Overview of application tasks

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

- Connecting to the data store
- Discovering the available metadata
- Selecting and calculating data through queries
- Gaining access to query results

The rest of this topic briefly describes these tasks, and the rest of this guide provides detailed information.

Task: Connecting to the data store

Before an application can connect to the data store, it must obtain the CORBA stub that resides on the OLAP API client computer and represents the OLAP service to which the connection will be made. The application can use a CORBA name-space browsing procedure outside the OLAP API to get the name of the OLAP service. With this name in hand, the application can obtain the stub. The application then uses the OLAP API `ConnectionManager` and `Connection` objects to specify the stub and establish the connection.

For more information about connecting, see Chapter 3.

Task: Discovering the available metadata

Having established a connection, the application creates an `MdmMetadataProvider`. This object gives access to all the metadata objects in the data store.

To discover the available metadata, an application uses the `getRootSchema` method on the `MdmMetdataProvider` to obtain the top-level measure folder for all of its metadata objects. The application then gets the dimensions, measures, and subfolders that are under the root. Once the application has all the dimensions and measures, it can interrogate them to get their attributes, hierarchies, levels, and other characteristics.

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. For information about how an application can discover the available metadata, see Chapter 4.

Task: Selecting and calculating 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 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.

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.

You can manipulate `Source` objects directly, using methods such as `select`, `remove`, and `appendValues` to create selections. In addition, you can use methods such as `plus`, `div`, and `total` to calculate values. `Source` and its subclasses, `NumberSource`, `StringSource`, and `BooleanSource`, have a rich assortment of methods for manipulating data. The most powerful method in `Source` is `join`, which gives you the ability to combine `Source` objects in almost any way imaginable.

If you are implementing a simple user interface, you might use only the methods on 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 use `Template` classes as described in the topic “Developing an OLAP API Application” on page 1-13. Within the code for each `Template`, you use the methods on 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 5. For information about working with `Template` objects, see Chapter 11.

Task: Gaining access to 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` based on the `Source` that specifies the query.

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 three `MdmDimension` objects. The `Cursor` for this `Source` has a structure that mirrors the `Source` itself; that is, the `Cursor` organization is based on the same three `MdmDimension` objects.

To retrieve all 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 `Cursor` objects to retrieve data, see Chapter 9.

2 Understanding OLAP API Metadata

Chapter summary

This chapter describes the metadata objects that the OLAP API provides, and explains how these objects relate to the metadata objects that a database administrator specifies when preparing the data in Oracle Enterprise Manager.

List of topics

This chapter includes the following topics:

Overview of the OLAP API Metadata

Multidimensional metadata model

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, hierachies, levels, and attributes. In this release, the OLAP API design incorporates an object-oriented model called MDM (multidimensional metadata). The OLAP API can also be extended to support other models.

The data in an Oracle database must be prepared by a database administrator in order to support the MDM model. Even though recent SQL enhancements have introduced some multidimensional objects, such as dimension, there are other objects and characteristics that must be added.

Data preparation

A database administrator starts with a data warehouse that is organized according to a star or snowflake schema. The schema specifies dimension tables as well as fact tables, which contain measures.

Using the OLAP management feature in Oracle Enterprise Manager, the administrator adds metadata to the data warehouse. The objects provided in this step supply the metadata required for OLAP Services to access the data. The metadata objects that are created in Oracle Enterprise Manager map to MDM metadata objects in the OLAP API.

Oracle Enterprise Manager is an administrative tool that is provided with Oracle. OLAP management is one of the choices offered within Oracle Enterprise Manager.

The topic,“OLAP Metadata Objects in Oracle Enterprise Manager” on page 2-2 briefly describes the metadata that a database administrator prepares for use with OLAP Services.

OLAP Metadata Objects in Oracle Enterprise Manager

Oracle Enterprise Manager’s OLAP management feature

Using the Oracle Enterprise Manager graphical user interface for OLAP management, a database administrator adds metadata to a data warehouse. The end result, within the graphical user interface, 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 objects maps directly to an MDM object in the OLAP API.

The collection of warehouse data for which a database administrator has created metadata using the OLAP management feature of Oracle Enterprise Manager is referred to as the data store to which the OLAP API gives access.

For detailed information about using the OLAP management feature of Oracle Enterprise Manager, see the Oracle Enterprise Manager Help system.

Note that the OLAP management feature includes a cube object which does not map directly to any MDM object. Database administrators use cubes in Oracle Enterprise Manager to specify the dimensions of each measure, as well as other characteristics. 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.

Database administrators also work with materialized views in the OLAP management feature of Oracle Enterprise Manager. These are relevant to query optimization, but they do not map to objects in the MDM model.

The rest of this topic briefly describes the OLAP management objects in Oracle Enterprise Manager that map directly to MDM objects in the OLAP API.

Dimensions in Oracle Enterprise Manager

The dimension property sheet in the OLAP management feature of Oracle Enterprise Managergives a database administrator the ability to specify the following for a given OLAP dimension.

- General characteristics, such as the name of the dimension and the schema from which its data is drawn.
- Levels, which record the levels of the dimension. The database administrator typically specifies one or more levels for each OLAP dimension.
- Hierarchies, which specify the parent-child relationships between the levels. The database administrator typically specifies at least one hierarchy for each OLAP dimension. If there is only one level for the dimension, then no hierarchy is specified and the dimension is a simple, non-hierarchical list.
- Attributes, which record characteristics of the level elements for the dimension. For example, an attribute might record the gender of each customer in the customers dimension.

- OLAP options, which record characteristics of the dimension or its levels, hierarchies, or attributes. For example, there are options for the plural name of the dimension, for its default hierarchy, and for the display names and descriptions for its levels, hierarchies, and attributes.

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. When database administrators are adding a dimension to a cube, they can specify an alias for the dimension.

Measures in Oracle Enterprise Manager

The cube property sheet in the OLAP management feature of Oracle Enterprise Manager gives a database administrator the ability to specify 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 measure's data resides. As an alternative, the database administrator can specify a calculation or transformation that produces the data.

Measure folders in Oracle Enterprise Manager

Once a database administrator has created measures (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 three separate folders for financials, sales, and human resources.

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, each measure's dimensions are included, along with its 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.

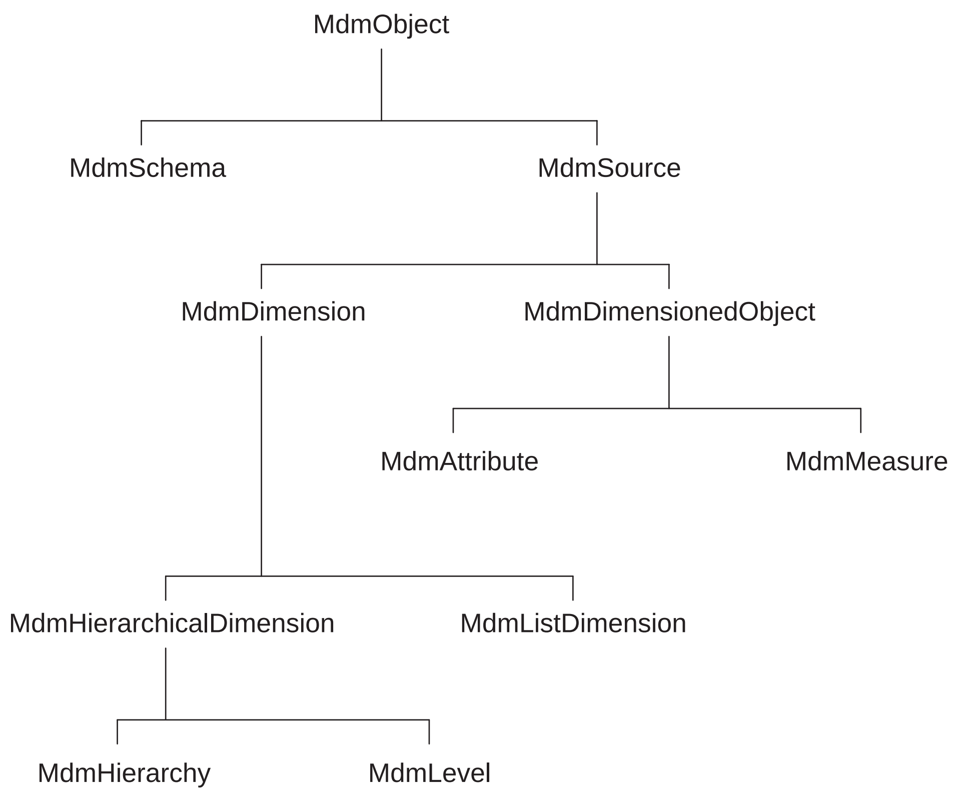
Overview of MDM Metadata Objects in the OLAP API

MDM Java classes

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

- The `MdmObject` class has the following subclasses: `MdmSchema` and `MdmSource`.
- The `MdmSource` class has the following subclasses: `MdmDimensionedObject` and `MdmDimension`.
- The `MdmDimensionedObject` class has the following subclasses: `MdmAttribute` and `MdmMeasure`.
- The `MdmDimension` class has the following subclasses: `MdmHierarchicalDimension` and `MdmListDimension`.
- The `MdmHierarchicalDimension` class has the following subclasses: `MdmHierarchy` and `MdmLevel`.

The diagram below shows this structure.



Mapping of Oracle Enterprise Manager objects to MDM objects

An application accesses 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 Oracle Enterprise Manager map directly to MDM metadata objects that are accessible through the `MdmMetadataProvider`. The following table presents the typical mapping.

Oracle Enterprise Manager Objects	MDM Metadata Objects
Dimension	`MdmHierarchy` or `MdmListDimension`
Hierarchy	`MdmHierarchy`
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.

MdmSchema class

An `MdmSchema` represents a set of data that is used for navigational purposes. An `MdmSchema` is a container for `MdmMeasure`, `MdmDimension`, 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 by a database administrator using the OLAP management feature of Oracle Enterprise Manager.

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 on the `MdmMetadataProvider`, as explained in Chapter 4.

The root `MdmSchema` contains all the `MdmMeasure` and `MdmDimension` objects that are in the data store. That is, if the root `MdmSchema` has subschemas that contain `MdmMeasure` and `MdmDimension` objects, the root `MdmSchema` also contains those objects.

An `MdmSchema` has methods for getting all the `MdmMeasure`, `MdmDimension`, and `MdmSchema` objects that it contains. The root `MdmSchema` also has a method for getting the measure MdmDimension, whose elements are all the MdmMeasure objects in the data store.

MdmSource class

An MdmSource represents a measure, dimension, or other set of data (such as an attribute) that is used for analysis. This abstract class is the basis for some important MDM metadata classes, such as MdmMeasure, MdmDimension, and MdmAttribute.

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.

In order to access the data values for a given MdmSource, an application calls the getSource method on 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 MdmDimension called mdmProductsDim.

```
Source productsDim = mdmProductsDim.getSource();
```

A Source that is the result of the getSource method on an MdmSource is called a primary Source. An application creates 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 5.

The rest of this chapter describes the subclasses of MdmSource, along with other classes, such as MdmDimensionDefinition and MdmDimensionMemberType, that are closely related.

MdmDimension Class

Description of an MdmDimension

An MdmDimension represents a list of elements that can organize a set of data. For example, if you have a set of sales figures for a given year and you organize them by month, the list of months is a dimension of the sales data. The values of the month dimension act as indexes for identifying each particular value in the set of sales data.

In the OLAP API, the abstract `MdmDimension` class represents the general concept of a list of elements that can organize data. `MdmDimension` has an abstract subclass called `MdmHierarchicalDimension`, which represents a list that has hierarchical characteristics.

The following concrete subclasses of `MdmDimension` represent the specific kinds of `MdmDimension` objects that can be used in analysis.:

- `MdmLevel`, which represents a list of elements that supply one level of a hierarchical structure. Each element can have a parent and one or more children. The parents and children of a given `MdmLevel` element are not within the given `MdmLevel`. They are elements of different `MdmLevel` objects.
- `MdmHierarchy`, which represents a list of elements arranged in a hierarchical structure that has levels based on parent-child relationships. Each element can have a parent and one or more children, and all of these elements are within the `MdmHierarchy`.

  Though the parent and child elements are within the `MdmHierarchy`, they correspond to elements in `MdmLevel` objects. Therefore, loosely speaking, an `MdmHierarchy` is composed of `MdmLevel` objects. Some `MdmHierarchy` objects are simply composed of `MdmLevel` objects. Others are unions of one or more subordinate `MdmHierarchy` objects, which in turn, are composed of `MdmLevel` objects.
- `MdmListDimension`, which represents a simple list of elements that play no part in any hierarchical structure. The elements have no parents and no children.

Both `MdmLevel` and `MdmHierarchy` are concrete subclasses of the abstract `MdmHierarchicalDimension` class.

An `MdmDimension` can have one or more `MdmAttribute` objects. Each of these objects maps the elements of the `MdmDimension` to values representing some characteristic of the elements. To obtain the `MdmAttribute` objects for a given `MdmDimension`, call its `getAttributes` method.

An `MdmDimension` has an `MdmDimensionDefinition`, which represents the structure of the underlying data, and an `MdmDimensionMemberType`, which represents the basic nature of the elements. These two objects hold important information about the `MdmDimension` to which they belong. For a given `MdmDimension`, you use its `getDefinition` and `getMemberType` methods to obtain these related objects.

Information held by an MdmDimensionDefinition

An `MdmDimensionDefinition` indicates the structure of the underlying data on which the `MdmDimension` is based. The `MdmDimensionDefinition` class is abstract. Therefore, instances are always one of the following subclasses:

- `MdmBaseDimensionDefinition`, which indicates that the `MdmDimension` has underlying data structured as a single list. For example, an `MdmLevel` is often based on a single column in a relational table.
- `MdmUnionDimensionDefinition`, which indicates that the `MdmDimension` has underlying data structured as the union of two or more lists. For example, an `MdmHierarchy` can be based on two or more columns in a relational table, one column for each `MdmLevel`.
- `MdmAliasDimensionDefinition`, which indicates that the `MdmDimension` acts as a proxy (that is, an alias) for another `MdmDimension`.

An `MdmDimension` that has an `MdmUnionDimensionDefinition` has regions. A region of a given `MdmDimension` is another `MdmDimension` that represents a subset of the elements of the given `MdmDimension`. For example, an `MdmDimension` for calendar year might have one region that represents quarters and another region that represents months. To obtain the regions of an `MdmDimension`, you call the `getRegions` method on its `MdmUnionDimensionDefinition`.

Information held by an MdmDimensionMemberType

An `MdmDimensionMemberType` indicates the basic nature of the elements in the `MdmDimension`. It holds a description for each element, and it often provides methods for finding out other information about individual elements. The `MdmDimensionMemberType` class is abstract. Therefore, instances are always one of the following subclasses:

- `MdmTimeMemberType`, which indicates that the `MdmDimension` elements represent time periods. An `MdmTimeMemberType` has methods for finding out the end date and time span for each element.
- `MdmMeasureMemberType`, which indicates that the `MdmDimension` elements are all the `MdmMeasure` objects in the data store. There is only one `MdmDimension` with an `MdmMeasureMemberType`, and it is referred to as the measure `MdmDimension`. You can obtain the measure `MdmDimension` by calling the `getMeasureDimension` method on the root `MdmSchema`.

- MdmStandardMemberType, which indicates that the MdmDimension elements have no specific characteristics. Most MdmDimension objects have an MdmStandardMemberType.

MdmLevel Class

Description of an MdmLevel

An MdmLevel is an MdmHierarchicalDimension whose parents and children are elements from other MdmLevel objects. The elements from a given MdmLevel correspond to a subset of the elements in an MdmHierarchy.

A given MdmLevel is based on a level that was specified by a database administrator in the OLAP management feature of Oracle Enterprise Manager. Typically, the database administrator specified a column in a database table to provide the elements for the level.

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 getParentRelation and getAncestorsRelation methods on the MdmLevel. Sometimes the parent and ancestors attributes are referred to as parent and ancestors relations.

Typically, an MdmLevel has an MdmBaseDimensionDefinition, because the underlying data is structured as a single list.

Elements of an MdmLevel

The list of elements in an MdmLevel includes only the elements in that one level. The values of the elements must be unique. However, uniqueness can be achieved by a database administrator who defines the level using two relational columns. For example, a level that represents cities can be defined in the relational database based on both the city column and the state column. This makes it possible for the value “Springfield” to appear for two different elements in the city level: one appears for Springfield, Illinois and another appears for Springfield, Massachusetts.

The following table lists the elements for an MdmLevel called mdmQuarter, which records the three-month quarters for a level MdmHierarchy called mdmTimesDimCalHier. This MdmHierarchy covers four years, so the number of elements in mdmQuarter is 16.

Elements of mdmQuarter
1998-Q1
1998-Q2
1998-Q3
1998-Q4
1999-Q1
1999-Q2
1999-Q3
1999-Q4
2000-Q1
. . .
2001-Q4

MdmHierarchy Class

Description of an MdmHierarchy

An MdmHierarchy is an MdmHierarchicalDimension that includes all the elements of one or more hierarchical structures. That is, all the parents and children are within the MdmHierarchy.

Even though the parent-child relationships exist in the MdmHierarchy, its elements are represented as a simple list. The relationships among the elements are recorded in the parent and ancestors attributes, which you can obtain by calling the getParentRelation and getAncestorsRelation methods on the MdmHierarchy. You can obtain the region for each element by calling the getRegionAttribute method on the MdmDimensionDefinition of the MdmHierarchy. Sometimes the parent, ancestors, and region attributes are referred to as parent, ancestors, and region relations.

Typically, an MdmHierarchy is one of the following two types:

- Level MdmHierarchy, which represents a hierarchical structure whose regions are MdmLevel objects. For example, a level MdmHierarchy for calendar year might have as its regions MdmLevel objects for year, quarter, month and day.

  A level MdmHierarchy has an MdmUnionDimensionDefinition, and its regions are MdmLevel objects. The return value from its getHierarchyType method is LEVEL_HIERARCHY. A level MdmHierarchy is based on a hierarchy that was defined by a database administrator in the OLAP management feature of Oracle Enterprise Manager.

- Union MdmHierarchy, which represents a dimension that has one or more subordinate hierarchical structures. These structures are represented by one or more level MdmHierarchy objects. An example, of an MdmHierarchy with two structures is a union MdmHierarchy for time that has two regions, one for the calendar year and another for the fiscal year. Each region is a level MdmHierarchy.

  A union MdmHierarchy has an MdmUnionDimensionDefinition and its regions are MdmHierarchy objects. The return value from its getHierarchyType method is UNION_HIERARCHY. A union MdmHierarchy is based on a dimension that was defined as having one or more hierarchies by a database administrator in the OLAP management feature of Oracle Enterprise Manager.

When working with level and union MdmHierarchy objects in the current release of the OLAP API, keep the following points in mind.

- Call the getAttributes method on a union MdmHierarchy, not on its subordinate level MdmHierarchy objects or their MdmLevel objects.
- Create queries on Source objects that are based on a level MdmHierarchy, not on a union MdmHierarchy.
- Call the getParentRelation and getAncestorsRelation methods on a level MdmHierarchy, not on a union MdmHierarchy.
- Call the getRegionAttribute method on the MdmUnionDimensionDefinition of a level MdmHierarchy, not of a union MdmHierarchy. This method returns the MdmAttribute that records the MdmLevel to which each MdmHierarchy element belongs.

Elements of a level MdmHierarchy

The elements of a level `MdmHierarchy` include all of the elements of all of its regions. The values of the elements in a particular level `MdmHierarchy` must be unique. The following examples present the elements of two level `MdmHierarchy` objects, one for calendar year and the other for fiscal year.

Example: A level MdmHierarchy for calendar year

The following table lists the elements for a level `MdmHierarchy` called `mdmTimesDimCalHier`, which includes the elements from four `MdmLevel` objects: `mdmYear`, `mdmQuarter`, `mdmMonth`, and `mdmDay`. The number of elements is 1529: 4 year elements, 16 quarter elements, 48 month elements, and 1461 day elements.

Elements of **`mdmTimesDimCalHier`**
1998
1998-Q1
1998-01
01-JAN-98
02-JAN-98
03-JAN-98
. . .
01-FEB-98
02-FEB-98
03-FEB-98
. . .
1998-Q2
1998-04
01-APR-98
02-APR-98
03-APR-98
. . .
1999

Elements of `mdmTimesDimCalHier`
1999-Q1
1999-01
01-JAN-99
02-JAN-99
03-JAN-99
. . .

Example: A level MdmHierarchy for fiscal year

The following table lists the elements for a level `MdmHierarchy` called `mdmTimesDimFisHier`, which includes the elements from four `MdmLevel` objects: `mdmFisYear`, `mdmFisQuarter`, `mdmFisMonth`, and `mdmFisDay`. The number of elements is 1529: 4 fiscal year elements, 16 fiscal quarter elements, 48 fiscal month elements, and 1461 fiscal day elements.

In this example, the `mdmFisDay` `MdmLevel` is based on the same relational database column on which the `mdmDay` `MdmLevel` is based (see the earlier example for calendar year). Therefore, the values of the elements for these two `MdmLevel` objects are identical. However, this does not mean that the elements themselves are identical. The elements in `mdmDay` are distinct from the elements in `mdmFisDay`; only the values of the two sets of elements are the same.

Elements of `timesDimFisHier`
FIS-1998
FIS-1998-Q1
FIS-1998-01
01-JUL-98
02-JUL-98
03-JUL-98
. . .
01-AUG-98
02-AUG-98

Elements of `timesDimFisHier`
03-AUG-98
. . .
FIS-1998-Q2
FIS-1998-04
01-OCT-98)
02-OCT-98
03-OCT-98
. . .
FIS-1999
FIS-1999-Q1
FIS-1999-01
01-JUL-99
02-JUL-99
03-JUL-99
. . .

Terminology: Nodes and leaves

A level `MdmHierarchy` represents a tree structure with parent-child relationships. Elements in the lowest `MdmLevel` are referred to as leaves, and the elements in the `MdmLevel` objects above the lowest level are referred to as nodes. Nodes have children; leaves do not.

Elements of a union MdmHierarchy

The elements of a union `MdmHierarchy` include all of the elements of all of its regions. Another way to say this is that a union `MdmHierarchy` includes all of the elements of all of the `MdmLevel` objects in all of its subordinate `MdmHierarchy` objects. In hierarchical terms, the set of elements includes all of the leaves (the elements at the lowest level) and all of the nodes (the elements at the levels above the lowest one) for all the hierarchies.

Distinct elements in the regions of a union MdmHierarchy

The elements in the regions of a union MdmHierarchy are totally distinct. That is, a given element does not appear in more than one region of a union MdmHierarchy. This is the case even if the database administrator in the OLAP management feature of Oracle Enterprise Manager specified the same level in two different hierarchies of a dimension. When this happens, OLAP Services creates two different MdmLevel objects, one for each level MdmHierarchy.

Though the elements of a union MdmHierarchy are distinct, the values of the elements are not required to be unique. Therefore in the example below, the leaf elements of the two regions of the union MdmHierarchy have values that are identical.

Example: A union MdmHierarchy for time

Consider a union MdmHierarchy called mdmTimesDim, which has two regions. The first region is the MdmHierarchy called mdmTimesDimCalHier, which has 1529 elements. The second region is the MdmHierarchy called mdmTimesDimFisHier, which also has 1529 elements. The set of elements for mdmTimesDim is the union of the elements from its two MdmHierarchy objects. Because no element can appear in both MdmHierarchy objects, mdmTimesDim has 3058 elements. Note that a calendar year begins on January 1, while a fiscal year begins on July 1.

The following table lists the elements of the union MdmHierarchy called mdmTimesDim. To distinguish the elements of mdmDay and mdmFisDay, whose values are identical, the word “(fiscal)” appears next to the values for mdmFisDay. The mdmDay and mdmFisDay objects were introduced earlier in the examples for the elements of a level MdmHierarchy.

Elements of mdmTimesDim
1998
1998-Q1
1998-01
01-JAN-98
. . .
1999
1999-Q1

Elements of `mdmTimesDim`
1999-01
01-JAN-99
. . .
FIS-1998
FIS-1998-Q1
FIS-1998-01
01-JUL-98 (fiscal)
. . .
FIS-1999
FIS-1999-Q1
FIS-1999-01
01-JUL-99 (fiscal)
. . .

MdmListDimension Class

Description of an MdmListDimension

An `MdmListDimension` is a simple lest of elements that have no hierarchical characteristics. That is, the notion of having a parent or a child is not relevant for the elements of an `MdmListDimension`.

Elements of an MdmListDimension

A given `MdmListDimension` is based on a dimension that was specified as having a single level and no hierarchy by a database administrator in the OLAP management feature of Oracle Enterprise Manager.

The following table lists the elements of an `MdmListDimension` called `mdmColor`.

Elements of `mdmColor`
Black
Blue
Cyan
Green
Magenta
Red
Yellow
White

MdmMeasure Class

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, the `MdmDimension` objects that organize an `MdmMeasure` provide the indexes for identifying individual cells.

For example, suppose you have an `MdmMeasure` for sales data, and the data is organized by product, time, customer, and channel (with channel representing the marketing method, such as direct or indirect.). You can think of the data as occupying a four-dimensional array with the product, time, customer and channel dimensions providing the organizational structure. The values of these four dimensions are indexes for identifying each particular cell in the array, which contains a single sales 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

A given `MdmMeasure` is based on an OLAP measure that was created by a database administrator in the OLAP management feature of Oracle Enterprise Manager. In most cases, the database administrator specified a column in a fact table to act as the basis for the OLAP measure (alternatively, the database administrator specified a mathematical calculation or a data transformation). In many but not all cases, the database administrator also specified at least one hierarchy for each of the measure’s OLAP dimensions, as well as an aggregation method. OLAP Services 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.

Typically, the `MdmDimension` objects of an `MdmMeasure` are union `MdmHierarchy` objects. That is, they have at least one hierarchical structure. It is important to remember that the elements of a union `MdmHierarchy` include all of the leaves and all of the nodes for all of the level `MdmHierarchy` objects that represent its regions. 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 management feature of Oracle Enterprise Manager. These values belong to `MdmMeasure` elements that are identified by a combination of leaf `MdmHierarchy` elements.
- Aggregated values that OLAP Services has provided. These values belong to `MdmMeasure` elements that are identified by at least one node element from an `MdmHierarchy`. The method for aggregation (for example, addition) was specified by the database administrator in the OLAP management feature of Oracle Enterprise Manager.

As an example, imagine an `MdmMeasure` called `mdmUnitCost` that is dimensioned by union `MdmHierarchy` objects called `mdmTimesDim` and `mdmProductsDim`. Each `MdmHierarchy` has leaf elements (for example, 01-JAN-99 in `mdmTimesDim`), and each `MdmHierarchy` has node elements (for example, 1999-Q1 in `mdmTimesDim`). A unique combination of two elements, one from each `MdmHierarchy`, 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 group 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, OLAP Services 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 OLAP Services makes available for analytical manipulation.

Example: An MdmMeasure with two MdmDimension objects

The table below lists some of the elements of the `MdmMeasure` called `mdmUnitCost`, which is described above. This `MdmMeasure` has `mdmProductsDim` and `mdmTimesDim` as its `MdmDimension` objects. Each of these objects is a union `MdmHierarchy` with regions that are level `MdmHierarchy` objects. For example, the level `MdmHierarchy` objects for `mdmTimesDim` are `mdmTimesDimCalHier` and `mdmTimesDimFisHier`, and the level `MdmHierarchy` for `mdmProductsDim` is `mdmProductsDimHier`.

Because there are so many elements in the `MdmMeasure`, the table shows only a few of them. For example, for `mdmTimesDim`, you should imagine that the ellipses (indicated by dots) cover additional days, months, quarters, and years in the `mdmTimesDimCalHier` region, as well as the entire `mdmTimesDimFisHier` region.

`mdmProductsDimHier` has three levels, which represent the product category (such as Boys), the product subcategory (such as Outerwear - Boys), and the individual product item (such as #23110). The table shows only one element from each level, and the ellipses cover all the rest.

Almost all the elements shown in the table are aggregated. The ones that are *not* aggregated are marked with an asterisk. These nonaggregated elements are the ones that are identified by the lowest level elements of both `mdmProductsDim` and `mdmTimesDim`.

Elements of `mdmProductsDim`	Elements of `mdmTimesDim`	Elements of `mdmUnitCost`
Boys	1998	12,800,444.00
Boys	1998-Q1	4,563,150.00
Boys	1998-01	1,837,254.00

Elements of `mdmProductsDim`	Elements of `mdmTimesDim`	Elements of `mdmUnitCost`
Boys	01-JAN-98	185,346.00
Boys	02-JAN-98	232,590.00
Boys	03-JAN-98	155,403.00
. . .	. . .	. . .
Outerwear -Boys	1998	6,473,065.00
Outerwear -Boys	1998-Q1	2,000,317.00
Outerwear -Boys	1998-01	637,482.00
Outerwear -Boys	01-JAN-98	27,009.00
Outerwear -Boys	02-JAN-98	20,346.00
Outerwear -Boys	03-JAN-98	12,498.00
. . .	. . .	. . .
23110	1998	847,362.00
23110	1998-Q1	200,635.00
23110	1998-01	60,735.00
23110	01-JAN-98	2,226.00 *
23110	02-JAN-98	1,709.00 *
23110	03-JAN-98	2,047.00 *
. . .	. . .	. . .

MdmAttribute Class

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. A typical example is an `MdmAttribute` that records the gender of each customer in an `MdmDimension` called `mdmCustomersDim`. In this case, the elements of the `MdmAttribute` have the values “Female” and “Male”.

The values of an `MdmAttribute` might be `String` values (such as “Female”), numeric values (such as 45), or objects (such as `MdmLevel` objects).

Like an MdmMeasure, an MdmAttribute has elements that are organized by its MdmDimension. For example, the gender MdmAttribute has one element (with “Female” or “Male” as its value) for each element of the MdmDimension called mdmCustomersDim.

Typically, not all of the elements of an MdmDimension have meaningful mappings to the values of a given MdmAttribute. For example, the gender MdmAttribute applies only to the lowest level of mdmCustomersDim, because gender makes no sense for higher levels such as cities or states. If an MdmAttribute does not apply to some elements of an MdmDimension, then their MdmAttribute values are 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.

Elements of an MdmAttribute

A given MdmAttribute is based on an attribute that was specified for a dimension or a level by a database administrator in the OLAP management feature of Oracle Enterprise Manager.

The following table lists the elements for an MdmAttribute called mdmCustomersDimGender, which is based on the MdmDimension called mdmCustomersDim. Note that the values of the MdmAttribute are null for the city, country, and region levels. There are meaningful values only for the customer level, where each customer is represented by a number.

Elements of mdmCustomersDim	Elements of mdmCustomersDimGender
. . .	. . .
Africa	null
South Africa	null
Cape Town	null
5420	Female
11650	Female
17880	Male

Elements of `mdmCustomersDim`	Elements of `mdmCustomersDimGender`
24120	Female
67720	Male
73960	Male
. . .	. . .

3

Connecting to a Data Store

Chapter summary

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

List of topics

This chapter includes the following topics:

- Overview of the Connection Process
- Connection Classes in the OLAP API
- Establishing a Connection
- Closing a Connection
- Interrupting a Connection

Overview of the Connection Process

Context for making a connection

Chapter 1 describes the software components that are involved when your application accesses data through the OLAP API. A review of the interaction among the components can be helpful in understanding what happens when your application establishes a connecton. The following steps describe a typical interaction among the components:

1. The application code makes a request for data by calling the OLAP API client software.

2. The OLAP API client software makes a call to the CORBA software, which passes the request to the OLAP service.
3. The OLAP service retrieves data from the Oracle database and performs the selections or calculations specified by the application.
4. The OLAP service fulfills the data request by passing the data to the application through the CORBA and OLAP API software.

What happens when you open a connection

When your application establishes a connection, the following events take place:

1. At the request of the application, the CORBA software establishes a connection between the application and the particular OLAP service that the application specified.
2. On behalf of the application, the OLAP service establishes a connection with its parent Oracle database instance. The connection is made with the user ID and password that were specified by the application. The scope of the data that is available to the application depends on the access rights assigned to the user ID.

Note that once the connection is established, your application can access the data store or it can access an analytic workspace, which is managed separately by the OLAP service. For more information about accessing an analytic workspace, see *Oracle9i OLAP Services Developer's Guide to the OLAP DML*.

Prerequisites for connecting

This chapter describes the OLAP API classes and methods that an application uses to establish a connection. Before attempting to use these classes and methods, you must ensure that your development environment is set up correctly. The development environment includes the application development computer, on which your OLAP API application will run, as well as the computer on which the OLAP service is running.

Ensure that your development environment meets the following requirements:

- The Oracle database and the OLAP service are running. Note that when the OLAP service is running, the CORBA software on the OLAP service computer is ready to accept connections.

- Your Oracle database user ID has access to the relational schemas on which the data store is based, and the user ID is set up to connect to the database through the OLAP service.
- The CORBA software is in place and ready to run on the application development computer.
- The OLAP API jar files are on the application development computer and are accessible to the application code.

See Appendix A for information about the OLAP API jar files and the CORBA software for the application development computer. For information about setting up a user ID to connect through an OLAP service, see the *Oracle9i OLAP Services Concepts and Administration Guide*.

Coding steps for making a connection

To make a connection, perform the following steps:

1. Get a CORBA stub that represents the OLAP service to which the connection will be made.
2. Create a `Properties` object and put into it all the connection parameters that are required. For example, include the Oracle database user ID and password.
3. Make the connection by calling a method on one of the OLAP API connection classes, which are described in "Connection Classes in the OLAP API" on page 3-4.

These steps are explained in more detail in the topic "Establishing a Connection" on page 3-7.

OLAP API classes involved in making a connection

You use the following OLAP API classes to make a connection:

- `ConnectionManager` class
- `Connection` class
- `ConnectionParameterInfo` class

These classes are described in more detail in the topic "Connection Classes in the OLAP API" on page 3-4. In addition to these OLAP API classes, you use CORBA classes, as described in "Step 1: Getting the CORBA stub for the first connect method parameter" on page 3-7.

Connection Classes in the OLAP API

Overview of the connection classes

The OLAP API connection classes are in the `oracle.express.connection` package. This topic focuses on the three important classes in the package: `ConnectionManager`, `Connection`, and `ConnectionParameterInfo`. One of the remaining classes, `Database`, is discussed briefly in Chapter 4.

ConnectionManager class

A `ConnectionManager` establishes a connection when you call its `connect` method. However, before you make this call, you must initialize the `ConnectionManager`.

To initialize a `ConnectionManager`, you use the `init` method in `ConnectionManager`. Because the method is overloaded, you can use either of its two forms. One form is appropriate for standalone Java applications. The other form is appropriate for Java applets.

- `init()`, which initializes the singleton `ConnectionManager` that is appropriate for use by all standalone Java applications. This `ConnectionManager` uses the singleton default ORB.
- `init(java.applet.Applet applet)`, which creates a new `ConnectionManager` that is appropriate for use by the specified applet. This `ConnectionManager` uses the ORB that was initialized with the applet.

For simplicity, the OLAP API documentation uses the term “application” to refer to both standalone applications and applets.

Once you have initialized the `ConnectionManager`, use the `connect` method in `ConnectionManager` to establish the connection. One of the parameters to the `connect` method is a java `Properties` object that holds the connection parameters (such as user name and password) for the connection.

A `ConnectionManager` can also provide information about the appropriate connection parameters through its `getConnectionParameterInfo` method. The reason that this method can be useful is because the parameters that are appropriate for connecting to one OLAP service might be different from the parameters that are appropriate for connecting to another OLAP service. If there is any reason to believe that more than the usual user ID and password parameters are required, then you can use the `getConnectionParameterInfo` method to find out what is needed.

The following table presents an overview of the methods in `ConnectionManager`.

Method	Return Value
init()	The singleton `ConnectionManager` for Java applications.
init(java.applet.Applet applet)	A new `ConnectionManager` for the specified Java applet.
getConnectionParameterInfo	A `List` of `ConnectionParameterInfo` objects, each of which represents a parameter that is relevant for making a connection to a particular OLAP service.
connect	A `Connection` object that represents a connection between an application and an OLAP service.
getOpenConnections	The `Set` of `Connection` objects that were created through this `ConnectionManager` and that are currently active.

For sample code that uses the `ConnectionManager` class, see “Step 2: Creating a Properties object for the second connect method parameter” on page 3-9 and “Step 3: Making the connection using the connect method” on page 3-10.

Connection class

A `Connection` object represents a connection between an application and an OLAP service, which is a child process of an Oracle database instance. You create a `Connection` object when you call the `connect` method on a `ConnectionManager`.

Most of the methods on `Connection` merely provide information about the OLAP service. For example, one method returns the name of the host computer, and another method returns the name of the OLAP service.

The following table presents the most important methods in the `Connection` class.

Method	Return Value
`close`	Closes the connection.

Method	Return Value
`isOpen`	A `boolean` value that indicates whether the connection is currently active.
`getDefaultDatabase`	A `Database` object that represents the Oracle database instance that is the parent of the OLAP service for the connection. Use this method when creating an `MdmMetadataProvider`, as described in Chapter 4.

For sample code that uses the `Connection` class, see "Step 3: Making the connection using the connect method" on page 3-10.

ConnectionParameterInfo

A `ConnectionParameterInfo` object holds information about a single parameter that can be used for making a connection.

Before making a connection, you can call the `getParameterInfo` method on the `ConnectionManager` in order to discover the connection parameters that are appropriate to the configuration of a particular OLAP service. The `getParameterInfo` method interrogates the OLAP service and returns a `List` of `ConnectionParameterInfo` objects, one for each appropriate parameter. You can then call the methods on each `ConnectionParameterInfo` object to find out the name of the parameter, whether it is required or optional, and the possible choices for the values of the parameter.

With this information, you can present choices to an end user through a graphical user interface, and collect the connection parameter information that is appropriate for that user. Having assembled the information to be specified, you can use the `put` method on a `Properties` object to store all the parameter names and values in the `Properties` object. Finally, you can pass the `Properties` object to the `connect` method on the `ConnectionManager`.

Discovering appropriate parameters for making a connection can be an iterative process. For example, you might call the `getConnectionParameterInfo` method once specifying an empty `Properties` object, but after you discover the properties that are relevant, you might make another call. This time you specify the `Properties` object populated with the parameters and values that were identified by the first call. Depending on the parameters and values specified, the second call to the `getConnectionParameterInfo` method might return a longer list of appropriate parameters that you can specify when making the connection.

The following table presents an overview of the methods on a `ConnectionParameterInfo` object. Each `ConnectionParameterInfo` object provides information about a single parameter.

Method	Return Value
`getName`	The name of the parameter.
`getDescription`	The description of the parameter.
`getChoices`	An array of `String` values that includes every valid value that an application can specify for the parameter.
`getValue`	The default value for the parameter.
`isRequired`	A `boolean` value that indicates whether the parameter is required.

For sample code that uses the `ConnectionParameterInfo` class, see “Step 2: Creating a Properties object for the second connect method parameter” on page 3-9.

Establishing a Connection

The connect method and its parameters

The communications link between an application and an OLAP service is through a CORBA implementation, as described in Chapter 1. The method that makes the connection is the `connect` method in `ConnectionManager`, and it requires the following parameters:

- The CORBA stub that identifies the OLAP service
- A `Properties` object that holds connection parameters

To establish a connection, complete the three steps described in this topic.

Note that the `connect` method is overloaded. There is an alternative version that accepts a third argument, which is the `Locale` for the connection. See the *Oracle9i OLAP Services OLAP API Reference* for details about this version of the `connect` method.

Step 1: Getting the CORBA stub for the first connect method parameter

The communications link between an application and an OLAP service is through a CORBA implementation, as described in Chapter 1. The method that makes the connection is the `connect` method on `ConnectionManager`, and it requires two parameters.

The first parameter to the connect method is a CORBA stub. This is a Java object that resides on the application computer and represents the OLAP service to which the connection will be made. Your application must use a CORBA naming service to obtain the stub. Oracle provides a CORBA naming service for use on most platforms. However, on some platforms the VisiBroker naming Service called Smart Agent is used. See the read me file for your installation of OLAP Services to find out which naming service is appropriate in your environment. If you will use Smart Agent, see Appendix B for coding information.

The following sample code for getting the stub uses the Oracle CORBA naming service and the Java Naming and Directory Interface (JNDI). The code uses the following three classes, which are in the Java Development Kit supplied by Sun Microsystems:

- `javax.naming.InitialContext`
- `javax.naming.Context`
- `java.util.Hashtable`

In addition, the code uses a constant from the `ServiceCtx` class, which resides in an Oracle CORBA naming service `jar` file that is supplied with your Oracle installation. See Appendix A for details.

The sample code specifies the following information:

- The URL that locates the target Oracle database for the connection. The URL has the following components, which are separated by colons.

  The prefix `sess_iiop`

  The host name. In the sample code, it is `//lab1.us.oracle.com`

  The listener port number for IIOP services. In the sample code, it is `2481`.

  The system identifier (SID) for the Oracle database instance. In the sample code, it is `bm1212`.

- The object name for the OLAP service that will accept the connection. The name includes its directory name in the namespace and the name of the published object. In the sample code, it is `/BI/OLAPService`.
- The user ID for the database connection. In the sample code, it is `hepburn`.
- The password for the database connection. In the sample code, it is `tracey`.

You can find out the information to specify for your own environment by talking to the OLAP Services database administrator. For detailed information about the syntax of the specifications, see the *Oracle CORBA Developer's Guide and Reference*.

```
import org.omg.CORBA.Object;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

String serviceURL = "sess_iiop://lab1.us.oracle.com:2481:bm1212";
String objectName = "/BI/OLAPService";
String useridValue = "hepburn";
String passwordValue = "tracey";

// Make hashtable to hold environment parameters for initial context
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, useridValue);
env.put(Context.SECURITY_CREDENTIALS, passwordValue);
env.put(Context.SECURITY_CREDENTIALS, ServiceCtx.NON_SSL_LOGIN);

// Get the CORBA stub for the OLAP service
javax.naming.Context ic = new InitialContext(env);
org.omg.CORBA.Object serviceStub = (org.omg.CORBA.Object)
       (ic.lookup(serviceURL + objectName));
```

Step 2: Creating a Properties object for the second connect method parameter

The second parameter to the `connect` method on `ConnectionManager` is a Java `Properties` object that holds the parameters for the connection.

The following sample code adds user ID and password parameters to a `Properties` object called `connParams`. Note that the access privileges that are granted to the user ID determine the scope of the data that will be visible in the application.

```
Properties connParams = new Properties();
String useridName = "UserID";
String useridValue = "hepburn";
String passwordName = "Password";
String passwordValue = "tracey";
connParams.put(useridName, useridValue);
connParams.put(passwordName, passwordValue);
```

If an application must prompt the user for information to be used as connection parameters, it can call the `getConnectionParameterInfo` method on `ConnectionManager`. This method returns a list of `ConnectionParameterInfo` objects, each of which describes one parameter. Then, the application can call methods on the `ConnectionParameterInfo` objects to get information about the parameters. Using this information, the application can prompt the user to specify parameter values for the connection, and the application can fill in the `Properties` object appropriately.

The following sample code prints the name and description for each parameter, just to illustrate how to work with a list of ConnectionParameterInfo objects.

```
ConnectionManager cm = ConnectionManager.init();
List cpiList = cm.getConnectionParameterInfo(serviceStub, new Properties());
Iterator conIt = cpiList.iterator();
System.out.println("Connection Parameters:");
while (conIt.hasNext()) {
    ConnectionParameterInfo cpi = (ConnectionParameterInfo) conIt.next();
    System.out.println("\tName : " + cpi.getName() + "\tDescription : " +
                       cpi.getDescription());
}
```

Step 3: Making the connection using the connect method

To make a connection, you initialize a `ConnectionManager` and call the `connect` method on it. When calling the `connect` method, you pass the following two parameters, which are described earlier in this topic:

- The CORBA stub (called `serviceStub` in earlier code examples)
- The `Properties` object that holds the connection parameters (called `connParams` in an earlier code example)

The following lines of code make a connection for a standalone application.

```
ConnectionManager cm = ConnectionManager.init();
Connection conn = cm.connect(serviceStub, connParms);
```

Closing a Connection

Using the close method

When you have completed your work with the data store, use the `close` method on the `Connection` object. In the following sample code, the `Connection` object is called conn.

```
conn.close();
```

What happens when you close a connection

When you close a connection, the following events take place:

1. The OLAP service terminates the connection that it made on behalf of your application with the Oracle database instance.
2. The CORBA software terminates the connection between your application and the OLAP service.

Interrupting a Connection

When to interrupt a connection

If an OLAP API method is taking too long to execute, and it is acceptable for your application to break off its connection with the OLAP service, you can use the `interrupt` method on `ConnectionInterrupter`. Terminating a connection in this way is not a routine procedure, and you should use it only under extraordinary circumstances.

Preparing to use the interrupt method

Typically, an application runs in a thread that establishes a connection and executes queries. Your application must prepare for using the `interrupt` method by creating a second thread for interrupting the connection. The second thread waits for notification by the first thread that an interruption is needed.

Using the interrupt method

When the first thread tells the second thread to terminate the connection, the second thread does the following:

1. Calls the `getRemoteStub` method on the `Connection` object that represents the connection to be interrupted. The method returns the CORBA stub for the `Connection`.
2. Uses the constructor to create a new `ConnectionInterrupter`, specifying the CORBA stub for the `Connection` as a parameter.
3. Calls the `interrupt` method on the new `ConnectionInterrupter` to terminate the connection.

4 Discovering the Available Metadata

Chapter summary

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

List of topics

This chapter includes the following topics:

Overview of the Procedure for Discovering Metadata

Scope of the available metadata: the data store

The OLAP API provides access to a collection of Oracle data for which a database administrator has created metadata using the OLAP management feature of Oracle Enterprise Manager. This collection of data and metadata is the data store for the application. Both the data and the metadata reside in the Oracle database instance.

Potentially, the data store includes all of the measure folders that were created by the database administrator in the OLAP management feature of Oracle Enterprise Manager. However, the scope of the data store that is visible when a given 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 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 on which the measures and dimensions are based.

MDM metadata

When the database administrator created the metadata, Oracle Enterprise Manager made measures, dimensions, and other OLAP objects. In the OLAP API, these objects are accessed as multidimensional metadata (MDM) objects, as described in Chapter 2. The mapping between the OLAP objects from Oracle Enterprise Manager and the MDM objects is automatically performed by OLAP Services.

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 the OLAP service and its parent Oracle database, as described in Chapter 3. Then, your application performs 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 characteristics of each `MdmMeasure` and `MdmDimension`. For example, for each `MdmMeasure` get its `MdmDimension` objects, and for each

MdmDimension find out whether it is a union MdmHierarchy , a level MdmHierarchy, an MdmLevel, or an MdmListDimension.

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

Discovering metadata and making queries

After you discover the metadata, you typically go on to create queries for selecting, calculating, and otherwise manipulating the data. In order to work with data in these ways, you must get the Source objects that OLAP Services has created to represent the data for querying. These Source objects are referred to as primary Source objects.

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

Function of an MdmMetadataProvider

An MdmMetadataProvider gives access to the metadata in a data store. It maps the metadata objects, such as measures, dimensions, and measure folders, that a database administrator created in the OLAP management feature of Oracle Enterprise Manager to the corresponding MDM objects, such as MdmMeasure, MdmDimension, and MdmSchema.

Creating some preliminary objects

Before you can create an MdmMetadataProvider, you must do the following:

- Create a TransactionProvider, which is required for constructing a DataProvider.
- Create a DataProvider, which is required for constructing an MdmMetadataProvider.
- Get the default Database, which is required for constructing an MdmMetadataProvider.

TransactionProvider is an interface, and DataProvider is an abstract class. Therefore, in your code, you use instances of the concrete classes called ExpressTransactionProvider and ExpressDataProvider.

The following code creates the preliminary objects on a Connection called conn. Chapter 3 explains how to create a Connection.

```
ExpressTransactionProvider tp = new ExpressTransactionProvider();
ExpressDataProvider dp = new ExpressDataProvider(conn, tp);
Database db = conn.getDefaultDatabase();
```

The TransactionProvider and DataProvider objects that are created 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 on this DataProvider object.

Creating the MdmMetadataProvider

The following code creates an MdmMetadataProvider using the preliminary objects described earlier.

```
MdmMetadataProvider mp = new MdmMetadataProivder (db, dp);
```

Getting the Root MdmSchema

Function of the root MdmSchema

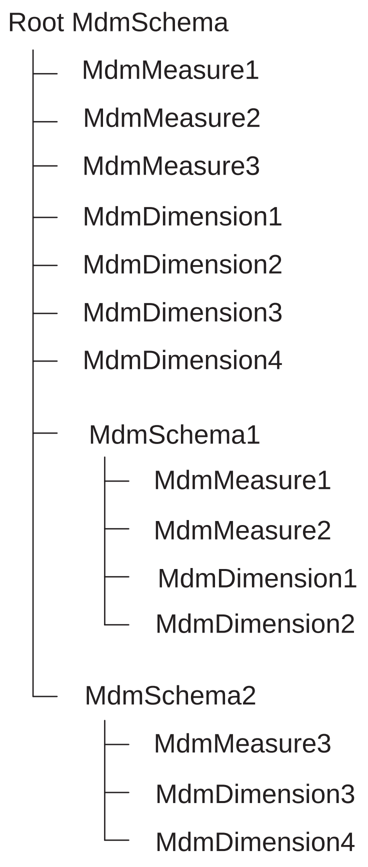
The metadata objects that are accessible through a given MdmMetadataProvider are organized in a tree-like structure, with the root MdmSchema at the top. Under the root MdmSchema are MdmMeasure objects, MdmDimension objects, and one or more MdmSchema objects, which are referred to as subschemas.

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

The root MdmSchema contains all the MdmMeasure and MdmDimension objects that are in the subschemas. Therefore, a given MdmMeasure or MdmDimension always appears twice in the tree. It appears once under the root MdmSchema and again under the subschema.

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 example in which one subschema has two MdmMeasure objects and two MdmDimension objects. Another subschema has one MdmMeasure object and two

MdmDimension objects. The root MdmSchema contains, in addition to the two subschemas, all three MdmMeasure objects and all four MdmDimension objects.



Using the OLAP management feature of Oracle Enterprise Manager, a database administrator arranges dimensions and measures under one or more top-level measure folders. When OLAP Services maps the measure folders to MdmSchema objects, it always creates the root MdmSchema above the MdmSchema objects for the top-level measure folders. Therefore, even if the database administrator creates only one measure folder, its corresponding MdmSchema will be a subschema under the root.

For more information about MDM metadata objects and how they map to OLAP objects in the OLAP management feature of Oracle Enterprise Manager, see Chapter 2.

Calling the getRootSchema method

The following code gets the root `MdmSchema` for an `MdmMetadataProvider` called mp.

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

Getting the Contents of the Root MdmSchema

MdmSchema contents

The root `MdmSchema` contains `MdmMeasure`, `MdmDimension`, and `MdmSchema` objects. In addition, the root `MdmSchema` has a measure `MdmDimension` that lists all the `MdmMeasure` objects.

Calling the getMeasures method

The following code gets a `List` of `MdmMeasure` objects that are in an `MdmSchema` called schema.

```
List measures = schema.getMeasures();
```

Calling the getDimensions method

The following code gets a `List` of `MdmDimension` objects that are in the `MdmSchema` called schema.

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

Calling the getSubSchemas method

The following code gets a `List` of `MdmSchema` objects that are in the `MdmSchema` called schema.

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

Calling the getMeasureDimension method

The following code gets the measure `MdmDimension` that is in the root `MdmSchema`. Use this method only on the root `MdmSchema`. It makes no sense to use it on subschemas, because only the root `MdmSchema` has a measure `MdmDimension`.

```
MdmMeasureDimension mdmMeasureDim = root.getMeasureDimension();
```

Getting the contents of subschemas

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

Getting the Characteristics of Metadata Objects

Getting the MdmDimension objects for an MdmMeasure

A primary characteristic of an `MdmMeasure` is that it has related `MdmDimension` objects. The following code gets a `List` of `MdmDimension` objects for an `MdmMeasure` called sales.

```
List dimsOfSales = mdmSalesAmount.getDimensions();
```

The `getMeasureInfo` method in the sample code provided later in this chapter shows one way to iterate through the `MdmDimension` objects belonging to a given `MdmMeasure`.

Getting the related objects for an MdmDimension

An `MdmDimension` has related `MdmDimensionDefinition` and `MdmDimensionMemberType` objects, which you can obtain by calling its `getDefinition` and `getMemberType` methods. If it is an `MdmHierarchy`, it also has regions, which you can obtain by calling the `getRegions` method on its `MdmUnionDimensionDefinition`.

The `getDimInfo` method in the sample code provided later in this chapter shows one way to get the following metadata objects for a given `MdmDimension`:

- Its `MdmDimensionMemberType`
- Its `MdmAttribute` objects

- Its concrete class and hierarchy type
- Its parent, ancestors, and region attributes
- Its `MdmDimensionDefinition`
- Its regions. That is, if it is a union `MdmHierarchy`, the code obtains its component `MdmHierarchy` objects. If it is a level `MdmHierarchy`, the code objains its component `MdmLevel` objects
- Its default level `MdmHierarchy`, if it is union `MdmHierarchy`.

Methods are also available for obtaining other `MdmDimension` characteristics. See the *Oracle9i OLAP Services OLAP API Reference* for descriptions of all the methods on the MDM classes.

Getting the Source for a Metadata Object

Difference between a metadata object and its Source

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 given metadata object, an application gets the `Source` object that represents its data. A `Source` that represents the data for a metadata object is called a primary `Source`.

Calling the getSource method

To get the primary `Source` for a metadata object, an application calls the `getSource` method on that metadata object. For example, if an application needs to display the sales figures for 1999, it must first use the `getSource` method on the `MdmMeasure` called `mdmSalesAmount`.

```
Source salesAmount = mdmSalesAmount.getSource();
```

An application can call the `getSource` method on any object that is an instance of a concrete subclass of `MdmSource`. The following is a list of the concrete subclasses:

- `MdmHierarchy`
- `MdmLevel`
- `MdmListDimension`

- `MdmAttribute`
- `MdmMeasure`

For more information about getting and working with primary `Source` objects, see Chapter 5

Sample Code for Discovering Metadata

Description of the sample code

The sample code that follows is a simple Java program called `SampleMetadataDiscoverer`. The program discovers the metadata objects that are under the root `MdmSchema` of any data store. The program's output lists the names and related objects for the `MdmMeasure` and `MdmDimension` objects in the root `MdmSchema` and its subschemas.

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 Sales History relational schema, which is provided with the installation of OLAP Services. In the OLAP management feature of Oracle Enterprise Manager, the Sales History schema is represented as the SH_CAT measure folder. Through an OLAP API connection, the SH_CAT measure folder maps to an `MdmSchema` that is also called `SH_CAT`.

The `SampleMetadataDiscoverer` program includes one piece of code that is specific to the `SH_CAT MdmSchema`. This code gets the primary `Source` for an `MdmDimension` for which the return value of the `getName` method is PRODUCTS_ DIM.

In most cases, an application will not search for a metadata object using its internal name (such as PRODUCTS_DIM), and it will not use the `System.out.println` method to produce output. However, this sample code uses these techniques because they offer the advantage of simplicty.

SampleMetadataDiscoverer program

To establish a connection, this program calls a hypothetical method called `connectOnLab1` on a hypothetical class called `MyConnection`. To close the connection, the program calls a method called

`MyConnection.closeConnection`. The code for these methods is not shown here, but the procedure for connecting is described in Chapter 3.

```
package mytestpackage;

import com.sun.java.util.collections.ArrayList;
import com.sun.java.util.collections.List;
import com.sun.java.util.collections.Iterator;

import oracle.express.connection.Connection;
import oracle.express.connection.Database;
import oracle.express.mdm.*;
import oracle.olapi.metadata.MetadataObject;

import oracle.olapi.data.source.Source;
import oracle.express.olapi.data.full.ExpressDataProvider;

public class SampleMetadataDiscoverer {

    static final int TERSE = 0;
    static final int VERBOSE = 1;

  public SampleMetadataDiscoverer(){
    }

  public static void main(String[] args) {

    // Connect to the service
    Connection conn = MyConnection.connectOnLab1();

    // Get the default database
    Database db = conn.getDefaultDatabase();

    // Create an MdmMetadataProvider
    MdmMetadataProvider mp =
      MyConnection.createMetadataProvider(conn, db);

    // Get metadata info about the root MdmSchema and its subschemas
    MdmSchema root = null;
    try {
        root = mp.getRootSchema();
        System.out.println("***Root MdmSchema: " + root.getName());
        MdmDimension measureDim = root.getMeasureDimension();
        System.out.println("******Measure MdmDimension: " +
           measureDim.getName());
        getSchemaInfo(root, TERSE);
    } catch (Exception e) {
        System.out.println("***Exception encountered : " + e.toString());
    }

    // Make a Source object out of the PRODUCTS_DIM MdmDimension
    System.out.println("***Making a Source object for PRODUCTS_DIM");

    MdmDimension productDim = null;
    try {
        List rootDims = root.getDimensions();
        Iterator rootDimIter = rootDims.iterator();
        while (productDim == null && rootDimIter.hasNext()) {
           MdmDimension aDim = (MdmDimension) rootDimIter.next();
           if (aDim.getName().equals("PRODUCTS_DIM"))
             productDim = aDim;
        }
        Source product = productDim.getSource();
        System.out.println("******Made the Source");
    } catch (Exception e) {
        System.out.println("******Exception encountered : " + e.toString());
    }

    // Close the connection
    MyConnection.closeConnection(conn);
  }

// *********************************************************

  // Method for getting info about an MdmSchema
  public static void getSchemaInfo(MdmSchema schema, int outputStyle) {

    System.out.println("***Schema: " + schema.getName());
    // Get the MdmSchema's dimension info
    MdmDimension oneDim = null;
    try {
        List dims = schema.getDimensions();
        Iterator dimIter = dims.iterator();
        System.out.println("   ");
        System.out.println("********************************************");
        System.out.println("   ");
       while (dimIter.hasNext()) {
           oneDim = (MdmDimension) dimIter.next();
           getDimInfo(oneDim, outputStyle);
           System.out.println("   ");
           System.out.println("********************************************");
```

```
                System.out.println("   ");
                }
          } catch (Exception e) {
              System.out.println("******Exception encountered : " + e.toString());
         }

         // Get the MdmSchema's measure info
         MdmMeasure oneMeasure = null;
         try {
             List measures = schema.getMeasures();
             Iterator measIter = measures.iterator();
             while (measIter.hasNext()) {
                 oneMeasure = (MdmMeasure) measIter.next();
                 getMeasureInfo(oneMeasure, outputStyle);
                 System.out.println("   ");
                 System.out.println("   ");
             }
          } catch (Exception e) {
              System.out.println("******Exception encountered : " + e.toString());
         }

         // Get the MdmSchema's subschema info
         MdmSchema oneSchema = null;
         try {
             List subSchemas = schema.getSubSchemas();
             Iterator subSchemaIter = subSchemas.iterator();
             while (subSchemaIter.hasNext()) {
                oneSchema = (MdmSchema) subSchemaIter.next();
                   getSchemaInfo(oneSchema, VERBOSE);
             }
         } catch (Exception e) {
             System.out.println("***Exception encountered : " + e.toString());
         }
       }

     // ********************************************************

       // Method for getting info about an MdmDimension
       public static void getDimInfo(MdmDimension dim, int outputStyle) {

         System.out.println("******MdmDimension Name: " + dim.getName());
         System.out.println("*********Description: " + dim.getDescription());

         if (outputStyle == VERBOSE) {
```

```
// Get MdmDimensionMemberType for the MdmDimension
try {
MdmDimensionMemberType dimMemberType = dim.getMemberType();
if (dimMemberType instanceof MdmStandardMemberType)
   System.out.println("*********Member Type: MdmStandardMemberType");
if (dimMemberType instanceof MdmTimeMemberType)
   System.out.println("*********Member Type: MdmTimeMemberType");
if (dimMemberType instanceof MdmMeasureMemberType)
   System.out.println("*********Member Type: MdmMeasureMemberType");
} catch (Exception e) {
    System.out.println("***Exception encountered : " + e.toString());
}

// Get attributes of the MdmDimension
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("***Exception encountered : " + e.toString());
}

// Get concrete class and hierarchy type of the MdmDimension
String kindOfDim = null;
try {
   if (dim instanceof MdmListDimension) {
      kindOfDim = "ListDim";
      System.out.println("*********" + dim.getName() +
               " is an MdmListDimension");
      }
   else if (dim instanceof MdmHierarchy)
      switch(((MdmHierarchy) dim).getHierarchyType()) {
         case (MdmHierarchy.UNION_HIERARCHY):
            kindOfDim = "UnionHier";
            System.out.println("*********" + dim.getName() +
               " is a union MdmHierarchy");
            break;
         case (MdmHierarchy.LEVEL_HIERARCHY):
            kindOfDim = "LevelHier";
            System.out.println("*********" + dim.getName() +
               " is a level MdmHierarchy");
            break;
         case (MdmHierarchy.VALUE_HIERARCHY):
            kindOfDim = "ValueHier";
            System.out.println("*********" + dim.getName() +
               " is a value MdmHierarchy");
            break;
     }
   else {
      kindOfDim = "Level";
      System.out.println("*********" + dim.getName() + " is an MdmLevel");
      }
} catch (Exception e) {
    System.out.println("***Exception encountered : " + e.toString());
}

// For level MdmHierarchy, get parent, ancestors, and region attributes
  if (kindOfDim.equals("LevelHier"))
   {
   System.out.println("*********Parent attribute: " +
      ((MdmHierarchicalDimension) dim).getParentRelation().getName());
   System.out.println("*********Ancestors attribute: " +
      ((MdmHierarchicalDimension) dim).getAncestorsRelation().getName());
   System.out.println("*********Region attribute: " +
      ((MdmUnionDimensionDefinition) dim.getDefinition())
      .getRegionAttribute().getName());
   }

// Get the MdmDimensionDefinition for the MdmDimension
MdmDimensionDefinition dimDef = dim.getDefinition();
// For union or level MdmHierarchy, list the regions and default hierarchy
if ((kindOfDim.equals("UnionHier")) || (kindOfDim.equals("LevelHier")))
   {
   try {
      System.out.println("   ");
      System.out.println("*********The following are the regions of " +
         dim.getName());
      List regions = ((MdmUnionDimensionDefinition)dimDef).getRegions();
      Iterator regIter = regions.iterator();
      while (regIter.hasNext()) {
         MdmDimension oneRegion = (MdmDimension) regIter.next();
         System.out.println("************" + oneRegion.getName());
       if (oneRegion.hasMdmTag(MdmMetadataProvider.DEFAULT_HIERARCHY_TAG))
         System.out.println("***************(The " + oneRegion.getName() +
              " region is the default MdmHierarchy)");
      }
   } catch (Exception e) {
       System.out.println("***Exception encountered : " + e.toString());
   }
}

// For union or level MdmHierarchy, get region info
if ((kindOfDim.equals("UnionHier")) || (kindOfDim.equals("LevelHier")))
   {
   try {
      System.out.println("   ");
      System.out.println("*********Information about the regions of " +
         dim.getName() + ":");
      List regions = ((MdmUnionDimensionDefinition)dimDef).getRegions();
      Iterator regIter = regions.iterator();
      while (regIter.hasNext()) {
         MdmDimension oneRegion = (MdmDimension) regIter.next();
         getDimInfo(oneRegion, VERBOSE);
         }
   } catch (Exception e) {
       System.out.println("***Exception encountered : " + e.toString());
   }
}
}
System.out.println("   ");
}

// *******************************************************

// Method for getting info about an MdmMeasure
public static void getMeasureInfo(MdmMeasure measure, int outputStyle) {
System.out.println("******Measure: " + measure.getName());
if (outputStyle == VERBOSE) {

// Get the dimensions of the MdmMeasure
try {
    List mDims = measure.getDimensions();
    Iterator mDimIter = mDims.iterator();
    while (mDimIter.hasNext())
       System.out.println("*********Dimension of the Measure: " +
          ((MdmDimension) mDimIter.next()).getName());
 } catch (Exception e) {
    System.out.println("******Exception encountered : " + e.toString());
}
}
}
}
```

Output from the SampleMetadataDiscoverer program

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

```
System.out.println("***Root MdmSchema: " + root.getName());
```

The code uses the `getName` method because its return value is brief. An alternative would be to use the `getDescription` method, but the output would be more verbose.

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

- The name of the root `MdmSchema`, which is ROOT.
- The name of the measure `MdmDimension` for the root `MdmSchema`. The name is MEASUREDIMENSION.
- The names and descriptions of the `MdmDimension` objects in the root `MdmSchema`.
- The names of the `MdmMeasure` objects in the root `MdmSchema`.
- Names, descriptions, and additional information about the `MdmDimension` and `MdmMeasure` objects in the SH_CAT `MdmSchema`.

  Because the `SH_CAT MdmSchema` is the only subschema under the root `MdmSchema`, its `MdmDimension` and `MdmMeasure` objects are identical to those in the root.
- Two lines that indicate that the code got the primary `Source` for the `MdmDimension` that has the name PRODUCTS_DIM.

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

```
***Root MdmSchema: ROOT
******Measure MdmDimension: MEASUREDIMENSION
***Schema: ROOT

********************************************
******MdmDimension Name: CHANNELS_DIM
*********Description: Channel Values

**********************************************
******MdmDimension Name: CUSTOMERS_DIM
*********Description: Customer Dimension Values

**********************************************
```

```
******MdmDimension Name: PRODUCTS_DIM
*********Description: Product Dimension Values

**********************************************
******MdmDimension Name: PROMOTIONS_DIM
*********Description: Promotion Values

**********************************************
******MdmDimension Name: TIMES_DIM
*********Description: Time Dimension Values

**********************************************

******Measure: SALES_AMOUNT

******Measure: SALES_QUANTITY

******Measure: UNIT_COST

******Measure: UNIT_PRICE

***Subschema: SH_CAT
***Schema: SH_CAT

********************************************

******MdmDimension Name: CHANNELS_DIM
*********Description: Channel Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********CHANNELS_DIM is a union MdmHierarchy

*********The following are the regions of CHANNELS_DIM
************CHANNEL_ROLLUP
***************(The CHANNEL_ROLLUP region is the default MdmHierarchy)

*********Information about the regions of CHANNELS_DIM:
******MdmDimension Name: CHANNEL_ROLLUP
*********Description: Standard Channels
*********Member Type: MdmStandardMemberType
*********CHANNEL_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION
```

```
*********The following are the regions of CHANNEL_ROLLUP
************CHANNEL_TOTAL
************CHANNEL_CLASS
************CHANNEL

*********Information about the regions of CHANNEL_ROLLUP:
******MdmDimension Name: CHANNEL_TOTAL
*********Description: Channel Total for the standard hierarchy
*********Member Type: MdmStandardMemberType
*********CHANNEL_TOTAL is an MdmLevel

******MdmDimension Name: CHANNEL_CLASS
*********Description: Channel Class level of the standard hierarchy
*********Member Type: MdmStandardMemberType
*********CHANNEL_CLASS is an MdmLevel

******MdmDimension Name: CHANNEL
*********Description: Channel level of the standard hierarchy
*********Member Type: MdmStandardMemberType
*********CHANNEL is an MdmLevel

**********************************************

******MdmDimension Name: CUSTOMERS_DIM
*********Description: Customer Dimension Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********Attribute: First Name
*********Attribute: Last Name
*********Attribute: Gender
*********Attribute: Marital Status
*********Attribute: Year of Birth
*********Attribute: Income Level
*********Attribute: Credit Limit
*********Attribute: Street Address
*********Attribute: Postal Code
*********Attribute: Phone Number
*********Attribute: E-mail
*********CUSTOMERS_DIM is a union MdmHierarchy

*********The following are the regions of CUSTOMERS_DIM
************GEOG_ROLLUP
***************(The GEOG_ROLLUP region is the default MdmHierarchy)
```

```
************CUST_ROLLUP

*********Information about the regions of CUSTOMERS_DIM:
******MdmDimension Name: GEOG_ROLLUP
*********Description: Standard
*********Member Type: MdmStandardMemberType
*********GEOG_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of GEOG_ROLLUP
************GEOG_TOTAL
************REGION
************SUBREGION
************COUNTRY
************STATE
************CITY
************CUSTOMER

*********Information about the regions of GEOG_ROLLUP:
******MdmDimension Name: GEOG_TOTAL
*********Description: Geography Total for the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********GEOG_TOTAL is an MdmLevel

******MdmDimension Name: REGION
*********Description: Region level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********REGION is an MdmLevel

******MdmDimension Name: SUBREGION
*********Description: Subregion level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********SUBREGION is an MdmLevel

******MdmDimension Name: COUNTRY
*********Description: Country level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********COUNTRY is an MdmLevel

******MdmDimension Name: STATE
*********Description: State level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********STATE is an MdmLevel
```

```
******MdmDimension Name: CITY
*********Description: City level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CITY is an MdmLevel

******MdmDimension Name: CUSTOMER
*********Description: Customer level of standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CUSTOMER is an MdmLevel


******MdmDimension Name: CUST_ROLLUP
*********Description: Standard
*********Member Type: MdmStandardMemberType
*********CUST_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of CUST_ROLLUP
************CUST_TOTAL
************STATE
************CITY
************CUSTOMER

*********Information about the regions of CUST_ROLLUP:
******MdmDimension Name: CUST_TOTAL
*********Description: Customer Total for the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CUST_TOTAL is an MdmLevel

******MdmDimension Name: STATE
*********Description: State level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********STATE is an MdmLevel

******MdmDimension Name: CITY
*********Description: City level of the standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
*********CITY is an MdmLevel

******MdmDimension Name: CUSTOMER
*********Description: Customer level of standard CUSTOMER hierarchy
*********Member Type: MdmStandardMemberType
```

```
*********CUSTOMER is an MdmLevel


**********************************************

******MdmDimension Name: PRODUCTS_DIM
*********Description: Product Dimension Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********PRODUCTS_DIM is a union MdmHierarchy

*********The following are the regions of PRODUCTS_DIM
************PROD_ROLLUP
***************(The PROD_ROLLUP region is the default MdmHierarchy)

*********Information about the regions of PRODUCTS_DIM:
******MdmDimension Name: PROD_ROLLUP
*********Description: Standard
*********Member Type: MdmStandardMemberType
*********PROD_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of PROD_ROLLUP
************PROD_TOTAL
************CATEGORY
************SUBCATEGORY
************PRODUCT

*********Information about the regions of PROD_ROLLUP:
******MdmDimension Name: PROD_TOTAL
*********Description: Product Total for the standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
*********PROD_TOTAL is an MdmLevel

******MdmDimension Name: CATEGORY
*********Description: Category level of standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
*********CATEGORY is an MdmLevel

******MdmDimension Name: SUBCATEGORY
*********Description: Sub-category level of standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
```

```
*********SUBCATEGORY is an MdmLevel

******MdmDimension Name: PRODUCT
*********Description: Product level of standard PRODUCT hierarchy
*********Member Type: MdmStandardMemberType
*********PRODUCT is an MdmLevel


**********************************************

******MdmDimension Name: PROMOTIONS_DIM
*********Description: Promotion Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********PROMOTIONS_DIM is a union MdmHierarchy

*********The following are the regions of PROMOTIONS_DIM
************PROMO_ROLLUP
***************(The PROMO_ROLLUP region is the default MdmHierarchy)

*********Information about the regions of PROMOTIONS_DIM:
******MdmDimension Name: PROMO_ROLLUP
*********Description: Standard Promotions
*********Member Type: MdmStandardMemberType
*********PROMO_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of PROMO_ROLLUP
************PROMO_TOTAL
************CATEGORY
************SUBCATEGORY
************PROMO

*********Information about the regions of PROMO_ROLLUP:
******MdmDimension Name: PROMO_TOTAL
*********Description: Promotions Total for the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
*********PROMO_TOTAL is an MdmLevel

******MdmDimension Name: CATEGORY
*********Description: Category level of the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
```

```
*********CATEGORY is an MdmLevel

******MdmDimension Name: SUBCATEGORY
*********Description: Sub-category level of the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
*********SUBCATEGORY is an MdmLevel

******MdmDimension Name: PROMO
*********Description: Promotion level of the standard PROMOTION hierarchy
*********Member Type: MdmStandardMemberType
*********PROMO is an MdmLevel


**********************************************

******MdmDimension Name: TIMES_DIM
*********Description: Time Dimension Values
*********Member Type: MdmStandardMemberType
*********Attribute: Long Description
*********Attribute: Short Description
*********Attribute: Period Number
*********Attribute: Period Number of Days
*********Attribute: Period End Date
*********TIMES_DIM is a union MdmHierarchy

*********The following are the regions of TIMES_DIM
************CAL_ROLLUP
***************(The CAL_ROLLUP region is the default MdmHierarchy)
************FIS_ROLLUP

*********Information about the regions of TIMES_DIM:
******MdmDimension Name: CAL_ROLLUP
*********Description: Calendar
*********Member Type: MdmStandardMemberType
*********CAL_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of CAL_ROLLUP
************YEAR
************QUARTER
************MONTH
************DAY
```

```
*********Information about the regions of CAL_ROLLUP:
******MdmDimension Name: YEAR
*********Description: Year level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********YEAR is an MdmLevel

******MdmDimension Name: QUARTER
*********Description: Quarter level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********QUARTER is an MdmLevel

******MdmDimension Name: MONTH
*********Description: Month level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********MONTH is an MdmLevel

******MdmDimension Name: DAY
*********Description: Day level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********DAY is an MdmLevel


******MdmDimension Name: FIS_ROLLUP
*********Description: Fiscal
*********Member Type: MdmStandardMemberType
*********FIS_ROLLUP is a level MdmHierarchy
*********Parent attribute: PARENTRELATION
*********Ancestors attribute: ANCESTORSRELATION
*********Region attribute: LEVELRELATION

*********The following are the regions of FIS_ROLLUP
************FIS_YEAR
************FIS_QUARTER
************FIS_MONTH
************FIS_WEEK
************DAY

*********Information about the regions of FIS_ROLLUP:
******MdmDimension Name: FIS_YEAR
*********Description: Year level of the Fiscal hierarchy
*********Member Type: MdmStandardMemberType
*********FIS_YEAR is an MdmLevel

******MdmDimension Name: FIS_QUARTER
*********Description: Quarter level of the Fiscal hierarchy
```

```
*********Member Type: MdmStandardMemberType
*********FIS_QUARTER is an MdmLevel

******MdmDimension Name: FIS_MONTH
*********Description: Month level of the Fiscal hierarchy
*********Member Type: MdmStandardMemberType
*********FIS_MONTH is an MdmLevel

******MdmDimension Name: FIS_WEEK
*********Description: Week level of the Fiscal hierarchy
*********Member Type: MdmStandardMemberType
*********FIS_WEEK is an MdmLevel

******MdmDimension Name: DAY
*********Description: Day level of the Calendar hierarchy
*********Member Type: MdmStandardMemberType
*********DAY is an MdmLevel

**********************************************

******Measure: SALES_QUANTITY
*********Dimension of the Measure: CHANNELS_DIM
*********Dimension of the Measure: CUSTOMERS_DIM
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: PROMOTIONS_DIM
*********Dimension of the Measure: TIMES_DIM

******Measure: SALES_AMOUNT
*********Dimension of the Measure: CHANNELS_DIM
*********Dimension of the Measure: CUSTOMERS_DIM
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: PROMOTIONS_DIM
*********Dimension of the Measure: TIMES_DIM

******Measure: UNIT_PRICE
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: TIMES_DIM

******Measure: UNIT_COST
*********Dimension of the Measure: PRODUCTS_DIM
*********Dimension of the Measure: TIMES_DIM

***Making a Source object for PRODUCTS_DIM
******Made the Source
```

5 Making Queries

Chapter summary

This chapter introduces `Source` objects which are the OLAP API objects that are the specifications for queries.

List of topics

This chapter includes the following topics:

How Does the OLAP API Represent Queries?

What objects represent queries?

In the OLAP API, queries are represented by two objects:

- The specification for a query is represented by a `Source` object. `Source` objects merely describe the data. They are not actual result sets.
- The result set of a query is a `Cursor` object. `Cursor` objects are the objects that you use to actually retrieve data from the database.

Only some of the query specifications represented by `Source` objects represent queries that the OLAP service can retrieve from a database and process. The kinds of `Source` objects for which you can define a `Cursor` and the use of `Cursor` objects are described more completely in Chapter 9.

`Source` objects are immutable. You cannot change a `Source` object once it has been created. When you want to present a `Source` object as changeable to your users (for example, to support what-if analysis), use a `Source` object defined by a `Template` object. `Template` objects themselves have state and can be modified at any time. For more information on using `Template` objects, see Chapter 11.

What are the subclasses of the Source class?

As outlined in the following table, the `Source` class has different subclasses for different data types. Each of the subclasses defines methods that are type-specific versions of various `Source` methods and methods that perform type-specific operations.

Class	Java Type of Element Values	OLAP API Data Type
`BooleanSource`	`boolean` values	Boolean
`DateSource`	Java `Date` objects	Date
`NumberSource`	`double`, `float`, `int`, or `short` values, or some combination of these numerical values	Double, Float, Integer, Short, or Number.
`StringSource`	Java `String` objects	String

For more information on these subclasses, see the online reference documentation for the OLAP API. For more information on OLAP API data type, see “Getting and Working with Fundamental Source Objects” on page 5-9.

What kinds of Source objects are there?

The OLAP API has the following kinds of `Source` objects:

- Primary `Source` objects which are `Source` objects that correspond to metadata objects. Primary `Source` objects have a structure that is similar to the metadata objects from which they are created.
- Derived `Source` objects which are new `Source` objects that are created by manipulating existing `Source` objects.
- Fundamental `Source` objects which are `Source` objects that represent data types and functions that are intrinsic to the OLAP API.

- Constant, list, and range Source objects which are simple nondimensional Source objects that you can use as operands when making selections and calculations.

Since a Source is an object, you must obtain an object reference to it in order to use it. The way you obtain an object reference to a Source varies by the kind of Source.

Getting Primary Source Objects

How to get primary Source objects

To get a primary Source object you take the following steps:

1. Create the metadata data object for which you want to create a corresponding Source object as described in Chapter 2.
2. Use the getSource method to create a Source object from the metadata object.

Example: Getting the Source for an MdmDimension

Assume that you have browsed through the metadata of a schema, identified an MdmDimension that has time values for both the fiscal year and the calendar year, and created an object named mdmTimesDim to represent it. To get the Source for this union dimension, you use the following syntax.

```
Source timesDim = mdmTimesDim.getSource();
```

Structure of a Source created from an MdmDimension

A primary Source that you create from an MdmDimension is a specification for a simple list of elements. This kind of Source does not have any keys itself, but it usual acts as a key to other Source objects. A primary Source created from an MdmDimension is called a nondimensional Source. You can think of it as a table with only a single column that holds the values of its elements.

Example: Structure of a nondimensional Source

In “Example: Getting a primary Source for an MdmMeasure” on page 5-4, we created a Source named timesDim from an MdmDimension named mdmTimesDim. The Source named timesDim has the same structure as mdmTimesDimCalHier that is illustrated in "Elements of a union MdmHierarchy" on page 2-16. It consists of a simple non-indexed list of elements.

Example: Getting a primary Source for an MdmMeasure

Assume that there is an MdmMeasure for which you have created an object named mdmUnitCost. To create a primary Source named unitCost for mdmUnitCost, you use the following code:

```
Source unitCost = mdmUnitCost.getSource;
```

Structure of a Source created from a MdmMeasure or an MdmAttribute

A primary Source that you create from an MdmMeasure or an MdmAttribute is a specification for a data set that has one or more keys. Each of these keys is a primary Source that was created from a MdmDimension. In other words, this kind of Source represents a set of data that is organized by one or more primary Source objects that have been created from MdmDimension objects.

You can conceptualize a primary Source created from an MdmMeasure or an MdmAttribute as a multidimensional array. The Source objects that were created from MdmDimension objects and that act as its keys are the dimensions of the array. The values of its dimensions are indexes for identifying each particular cell in the array, which contains a single value. You must specify a value for each dimension in order to identify a value in the array. Thus, the set of elements that are in a dimensional Source is determined by the structure of the Source objects that act as its keys.

In relational terms, you can also conceptualize a Source that you create from an MdmMeasure or an MdmAttribute as a table that has one column for its elements and one column for the elements of each of the Source objects that act as its keys. A Source object that is a key to another Source is often a primary key in a table in the underlying database. Consequently, when one Source is a key to another Source, you can think of the Source that is the key as a *foreign key*. When a Source has foreign keys, the *primary key* of the Source is a composite key (or multisegmented key) that consists of its foreign keys. Each element of one Source is identified by a set of elements of the Source objects that are its foreign keys.

The Source objects that act as the keys of a dimensional primary Source are known as inputs. An *input* is a foreign key to a Source object for which values have not yet been specified. A Source object that has an input knows the identity and characteristics of the input Source but does not know the values of the elements of the input. As a result, when a Source has inputs, the primary keys to its elements are not fully specified and the OLAP service cannot identify the elements of the Source. Thus, the query specification represented by a Source that has an input is incomplete. Consequently, you cannot create a Cursor on a primary Source and, therefore, you cannot retrieve its values into the application. To retrieve the values represented by a dimensional primary Source, you must derive a new Source from it by specifying elements for the values of the Source objects that act as its keys as described in “Selecting Elements Based on Key Values” on page 6-1.

Example: Structure of a dimensional Source

In “Example: Getting a primary Source for an MdmMeasure” on page 5-4, we created a primary Source named unitCost from the MdmMeasure named mdmUnitCost. The Source named unitCost has a structure that is similar to the structure of the MdmMeasure named mdmUnitCost illustrated in “Elements of an MdmMeasure” on page 2-19. It consists of elements that are indexed by the elements of productsDim and timesDim. The specification for the primary Source named unitCost does not include values for the Source objects that act as its keys (that is, productsDim and timesDim). In order to retrieve one or more elements of unitCost, you must specify the key values for productsDim and for timesDim that will uniquely identify the desired elements. For information on selecting key values, see “Selecting Elements Based on Key Values” on page 6-1.

Creating Derived Source Objects

How to create derived Source objects

You can derive new Source objects from existing Source objects by using the methods in the Source class and its subclasses or by using the generateSource method in the Template class. Template objects are extensions to the OLAP API that represent end-user concepts such as cubes, edges, and selections. They form a bridge between the requirements of the user interface and the powerful, but abstract, OLAP API logical model. Unlike other OLAP API objects, Template objects have state. Consequently, they can be modified at any time, even after they have been incorporated into some larger Source. The Source defined by a Template can be said to be dynamic in the sense that it can be changed. For more information about Template objects and how to define and work with Source objects within them, see Chapter 11.

Introducing the OLAP API Source methods

The OALP API includes primitive methods and shortcut methods.

Primitive methods

The primitive join method is the single most important Source creation method in the OLAP API. The primitive join method combines the elements of this Source (sometimes called the base Source) and another Source (called the joined Source) and filters this result set using a third Source (called the comparison Source) in the specified manner. Using an optional parameter, you can also use the primitive join method to add the joined Source as a dimension (or key) to the new Source. The primitive join method is discussed in more detail in "Introducing the join method" on page 5-7 and documented in detail the online reference documentation for the OLAP API.

The following table outlines the other primitive methods in the OLAP API.

Primitive Method	Description
alias	Creates a new Source object that is the same as the base Source object, but that has the base Source as its type.
distinct	Removes the duplicate rows (tuples) in this Source object.
extract	When the elements of the base Source are other Source objects, creates a new Source that has the base Source as an extraction input.
position	Creates a new Source object with the same structure as the base Source and whose elements are the position of the elements of the base Source.
value	Creates a new Source object that has the elements of the base Source and that has the base Source as an input.

These methods are documented in the online reference documentation provided for the OLAP API. For more information about using these methods to create derived Source objects, see Chapter 6 and Chapter 7.

Shortcut methods

The OLAP API provides various shortcut and convenience methods that you can use instead of the primitive join method. These methods include shortcuts for the primitive join method, as well as shortcut methods such as appendValue, at, cumulativeInterval, first, ge, interval, selectValues, and sortAscending.

These methods are documented in the online reference documentation provided for the OLAP API. For more information about deriving Source objects using these methods, see Chapter 6 and Chapter 7.

Introducing the join method

The most important primitive method in the OLAP API is the primitive `join` method.

Syntax: the primitive join method

The signature of the primitive `join` method is shown below:

```
Source join(Source joined,
            Source comparison,
            int comparisonRule,
            boolean visible)
```

The parameters are described below:

- `joined` is the `Source` that you want to join to this `Source`.
- `comparison` is the `Source` that you want to use as a filter for the join.
- `comparisonRule` is the rule that determines how the method uses the comparison `Source` to filter the result set. Specify a value for this parameter using one of the `Source.COMPARISON_RULE` fields.
    - `COMPARISON_RULE_SELECT` specifies that the new `Source` contains only those elements that appear in the comparison `Source`.
    - `COMPARISON_RULE_ASCENDING`, like `COMPARISON_RULE_SELECT`, specifies that the new `Source` contains only those elements that appear in the comparison `Source`, additionally, once the rows of the cross-product have been intersected by the comparison `Source`, the remaining rows are sorted by the value of the joined `Source` according to the position defined in the comparison `Source`.
    - `COMPARISON_RULE_DESCENDING`, like `COMPARISON_RULE_SELECT`, specifies that the new `Source` contains only those elements that appear in the comparison `Source`, additionally, once the rows of the cross-product have been intersected by the comparison `Source`, the remaining rows are sorted by the value of the joined `Source` according to the reverse position defined in the comparison `Source`.

  - COMPARISON_RULE_REMOVE specifies that the new Source created by a join contains only those elements that do not appear in the comparison Source.

- visible is a flag that specifies whether you want the joined Source object to be an output of the new Source. When true is specified, the joined Source becomes a dimension of the new Source and the values of the joined Source become the elements of that dimension.

The result of the primitive join method

The result of the join method is a new Source object. Depending on the complexity of the Source objects that you are joining, the resulting Source object may be simple or complex:

- When you join two nondimensional Source objects, the new Source is dimensioned by the joined Source and the new Source is simply the cross-product of the two Source objects.
- When you join dimensional Source objects, the new Source has the combined dimensionality of the base, joined, and comparison Source objects. Additionally, when you specify true for the value of the visible parameter, the joined Source becomes a dimension or key of the new Source. (For details on how the dimensions of the new Source are determined, see the online Help for the primitive join method.)

Example: Select option of join

Assume that you have a Source named myStates that does not have any inputs or outputs and whose elements are CA, MA, and NY and a Source named myProducts that does not have any inputs or outputs and whose elements are Dresses - Girls and Shirts - Girls. Now we issue the following code.

```
String[] values = new String[] {"NY", "CA"};
Source newSource = myProducts.join(myStates, values,
    Source.COMPARISON_RULE_SELECT, true);
```

When processing this code, the OLAP service takes the cross-product of myProducts and myStates, and then selects from the result only those rows for which the value of region is in the set of values {"NY", "CA"}. Another way of describing this processing is to say that the states output (column) is intersected with the comparison set {"NY", "CA"}.

This yields the result set shown below. Notice that the rows containing "MA" have been removed.

Input	Elements
states	**product**
CA	Dresses - Girls
CA	Shirts - Girls
NY	Dresses - Girls
NY	Shirts - Girls

Getting and Working with Fundamental Source Objects

What is Type in the OLAP API?

Type in the OLAP API is the set of `Source` elements from which a `Source` object obtains the values of its elements. You can retrieve the OLAP API type of a `Source` using the `getType` method that the `Source` class inherits from the `DataDescriptor` class.

When you create a new `Source` object by using a method on an existing `Source` object, typically the type of the new `Source` object is the base `Source`. In other words, the elements of the new `Source` object are obtained from the set of elements of the base `Source` object.

For example, assume that you have a `Source` object named `customer` whose elements are the unique numerical identifier for each customer. The OLAP API type of customer is Integer. Assume, additionally, that you use the `select` method on customer to create another `Source` object named `customerSelection`. The OLAP API type of `customerSelection` is `customer`.

What are the OLAP API data types?

The OLAP API data types and their relationship to each other are shown in the following table.

OLAP API Data Type	Description
Value	A `Source` object with any OLAP API data type.
Date	A Source object whose elements are Java `Date` objects.
Number	A `Source` object with any of OLAP API numerical data type.
Double	A `Source` object whose values have the Java `double` data type.
Float	A `Source` object whose values have the Java `float` data type.
Integer	A `Source` object whose values have the Java `int` data type.
Short	A `Source` object whose values have the Java `short` data type.
String	A Source object whose elements are Java `String` objects.
Boolean	A `Source` object whose values have the Java `boolean` data type.
Empty	A `Source` object that does not have any elements defined for it.
Null	A `Source` object that has a single element with the value of `null`.

Retrieving the OLAP API data type of a Source

You can retrieve the OLAP API data type of a `Source` using the `getDataType` method that the `Source` class inherits from the `DataDescriptor` class.

Also, because the OLAP API is an object-oriented API, it provides a `FundamentalMetadataObject` to represent each of the fundamental Java data types and the Java `String` object. These objects are known as the OLAP API data types. You can create a `Source` object that represents a `FundamentalMetadataObject`.

Creating Objects that Represent OLAP API Data Types

You can retrieve the objects that represent the OLAP API data types using methods on the `FundamentalMetadataProvider`. Each of these methods returns a

FundamentalMetadataObject. The OLAP API data types and the methods you use to retrieve them are shown in the following table.

OLAP API Data Type	Method That Retrieves This Data Type
Value	getValueDataType
Boolean	getBooleanDataType
Date	getDateDataType
Number	getNumberDataType
Double	getDoubleDataType
Float	getFloatDataType
Integer	getIntegerDataType
Short	getShortDataType
String	getStringDataType
Empty	getEmptyDataType To retrieve an empty Source, use the DataProvider.getEmptySource method.
Null	getVoidDataType To retrieve a null Source, use the DataProvider.getVoidSource method.

Steps: Creating a Source that represents an OLAP API data type

To create a Source object that represents an OLAP API data type, take the following steps:

1. Get the FundamentalMetadataProvider by using the getFundamentalMetadataProvider method on the DataProvider class.
2. Create the FundamentalMetadataObject object that represents the OLAP API data type by using the appropriate method on the FundamentalMetadataProvider class.
3. Create a Source from the objects returned in Step 1 by using the FundamentalMetadataObject.getSource method.

Example: Creating a Source for the OLAP API Boolean data type

The code shown below creates a Source object that represents the OLAP API Boolean data type.

```
FundamentalMetadataObject myFundamentalMetadataProvider =
    myDataProvider.getFundamentalMetadataProvider();
FundamentalMetadataObject olapBooleanFundObj =
    myFundamentalMetadataProvider.getBooleanType();
Source olapBooleanDataType = olapBooleanFundObj.getSource();
```

Now you can use olapBooleanDataType to check to see if the OLAP API data type of any other Source is Boolean.

Creating Constant, List and Range Source Objects

How to create constant, list, and range Source objects

You create simple nondimensional Source objects that you can use as operands by using the createConstantSource, createListSource, and createRangeSource methods on the DataProvider class.

Example: Creating a constant Source

Assume that you have an object named myDataProvider that represents the DataProvider used by your application and that, for computational purposes, you want a Source with a single element that has a value of 4. To create this Source you issue the following code

```
NumberSource myConstantFour = myDataProvider.createConstantSource(4);
```

6 Selecting Data

Chapter summary

This chapter discusses how to make data selections using the OLAP API.

List of topics

This chapter includes the following topics:

- Selecting Elements Based on Key Values
- Selecting Elements Based on Element Values
- Selecting Elements Based on Rank
- Selecting Elements Based on Hierarchical Position

Selecting Elements Based on Key Values

Why you need to specify values for the keys of a Source

As mentioned in "What objects represent queries?" on page 5-1, even though it helps to think of a `Source` object as a tabular or dimensional result set, a `Source` actually is *not* a result set. Instead, a `Source` object is a specification for a query that defines a result set. As part of this specification, a `Source` object keeps track of the keys for which values have been specified. Looking at keys from this point of view, you can say that a `Source` object has two different types of keys:

- *Inputs* which are keys for which values have not yet been specified. When a primary `Source` object has other `Source` objects that act as its keys, these `Source` objects are *always* inputs. Thus, the query specification represented by a dimensioned primary `Source` or any other `Source` that has an input is incomplete. You cannot create a `Cursor` for this type of `Source` and, consequently, you cannot retrieve the query specified by the `Source`.

- *Outputs* which are keys for which values have been specified. When a `Source` has only outputs, the primary key to its elements are fully specified. The query that this type of `Source` specifies is determinable. You can create a `Cursor` for this type of `Source` and use the `Cursor` to retrieve the data set specified by the `Source`.

If you want to create a `Cursor` on a `Source` object, all of the keys of the `Source` must be outputs. Consequently, to display a primary dimensional `Source`, you must first specify values for the keys of that `Source`. Specifying values for the keys of a `Source` is called *changing inputs to outputs.*

How to turn inputs into outputs?

The need to specify values for the keys of a dimensional `Source` with inputs is so universal, that the OLAP API has a `join` shortcut method to support it. To specify values for the keys of a dimensional `Source`, thereby changing an input to an output, use the following `join` method where the original `Source` is the `Source` object that has the input that you want to become an output and the joined `Source` is the input you want to change.

```
join (Source joined)
```

This is a shortcut for the following `join` method.

```
join (joined, emptySource, Source.COMPARISON_RULE_REMOVE, true);
```

Note that the comparison `Source` is the empty `Source` that has no elements. Consequently, even though the `COMPARISON_RULE_REMOVE` constant is specified, no elements are removed as a result of the comparison. Also, because the `visible` flag is set to `true`, the joined `Source` becomes an output of the new `Source`.

Additionally, since many of the methods of `Source` class and its subclasses are actually shortcut and convenience methods that implicitly call the `join` method, some of these methods also change inputs to outputs.

How does the structure of a dimensional Source determine its processing?

The way a dimensional `Source` is processed is determined by its structure. When a `Source` has both inputs and outputs, its elements (tuples) are identified by the set of its input and output values. In this case, each set of possible input values typically identifies a number of elements (tuples). Within this subset of data, the tuples are arranged by output. When a `Source` has inputs, many `Source` methods work on this subset of data.

For example, when a `Cursor` is opened on a `Source`, the OLAP service loops over its outputs in order to produce the data, but it (arbitrarily) qualifies away any of its inputs. Additionally, the OLAP service loops over the outputs of a `Source` when it processes any aggregation methods like `average` and `total`. In this sense, moving a `Source` from the list of inputs to the list of outputs is similar to moving a column out of the GROUP BY list in SQL.

For more information on the structure of a `Source` with inputs, see “Finding the position of elements” on page 6-7; for more information on fastest-varying and slowest-varying columns, see “What is the effect of input-output order on Source structure?” on page 6-3.

What is the effect of input-output order on Source structure?

The structure of a dimensioned `Source` is determined by the order in which you turn the inputs of the `Source` into outputs. The fastest-varying column is always the column that contains the elements of the `Source`. For a `Source` that has outputs, the first output that was created is the fastest-varying key column; the last output that was created is the slowest-varying key column.

When you string two `join` methods together in a single statement, the first `join` (reading left to right) is processed first. Consequently, when creating a single statement containing several `join` methods, make sure that the input that you want to be the fastest-varying of the new `Source` is the joined `Source` in the first `join` in the statement.

You can retrieve the inputs of a `Source` using the `getInputs` method that the `Source` class inherits from the `DataDescriptor` class. You can retrieve the outputs of a `Source` using the `getOutputs` method that the `Source` class inherits from the `DataDescriptor` class.

Example: Effect of input-output order on Source structure

Assume that you have a primary `Source` named `unitCost` that you created from a `MdmMeasure` object named `mdmUnitCost`. The `Source` named `unitCost` has inputs of `timesDim` and `productsDim`, and no outputs.The `timesDim` and `productsDim` `Source` objects do not have any inputs or outputs. The order in which you turn the inputs of `unitCost` into outputs determines the structure of a `Source` on which you can create a `Cursor`.

Joining first to timesDim

Assume also that you issue the following code to turn the inputs of the primary `Source` named `unitCost` into outputs.

```
Source newSource = unitCost.join(timesDim).join(productsDim);
```

This code strings two `join` methods together. Because `unitCost.join(timesDim)` is processed first, the key values for `timesDim` are the first key values specified. You can also say that `timesDim` is the first output defined for the new `Source`. After the first `join` is processed, the query specification represented by the resulting unnamed `Source` consists of the name of its input (that is, `productsDim`) and both the name and the element values of its output (that is, `timesDim`). You can think of the new unnamed `Source` as having the structure depicted below.

Output2	Output1	Element
`productsDim`	`timesDim`	`unit_Cost`
Boys		
...		
49780		
...		

After the second `join` is processed, the query specification represented by `newSource` consists of the names and the element values of both of its output (that is, `timesDim` and `productsDim`). Since `timesDim` was the first key for which values were specified, it is the fastest-varying output and the new `Source` has the structure depicted below.

Output2	Output1	Element
`productsDim`	`timesDim`	`unit_Cost`
Boys	1998	4,000
Boys	...	...
Boys	31-DEC-01	10
...	...	...
49780	1998	500

Output2	Output1	Element
49780	...	...
49780	31-DEC-01	9

Joining first to productsDim

Assume that you issue the following code to turn the inputs of unitCost into outputs.

```
Source newSource = unitCost.join(productsDim).join(timesDim);
```

This code strings two `join` methods together. Because `unitCost.join(productsDim)` is processed first, `productsDim` is the first output defined for the new `Source`. Consequently, `productsDim` is the fastest-varying output and the new `Source` has the structure depicted below.

Output2	Output1	Element
timesDim	productsDim	unitCost
1998	Boys	4,000
1998	...	...
1998	49780	500
...	...	...
31-DEC-01	Boys	10
31-DEC-01	...	...
31-DEC-01	48780	9

Selecting Elements Based on Element Values

How you select elements based on value

Typically, you use one of the following shortcut methods to select one or more elements in a `Source` based on its value:

- `Source.select(BooleanSource)`, `Source.selectValue(Source)`, `Source.selectValues(Source)`, and `Source.selectValues(Source[])` methods.

- `BooleanSource.selectValue(boolean)` and `BooleanSource.selectValues(boolean[])` methods.
- `NumberSource.selectValue(double)`, `NumberSource.selectValue(int)`, `NumberSource.selectValue(float)`, and `NumberSource.selectValue(short)` methods and the `NumberSource.selectValues(double[])`, `NumberSource.selectValues(float[])`, `NumberSource.selectValues(int[])`, and `NumberSource.selectValues(short[])` methods.
- `StringSource.selectValue(String)` and `StringSource.selectValues(String[])` methods.

You can also select elements using the primitive `join` method by using the `COMPARISON_RULE_SELECT` constant as shown below.

```
Source Source::join (Source joined,
     Source comparison,
     Source.COMPARISON_RULE_SELECT,
     boolean visible)
```

Example: Selecting based on element values

Assume that you have a primary `Source` objects named `timesDim` that you created from an `MdmDimension` object named `mdmTimesDim` and whose elements are the calendar values.

To select only the those values for 1996, you can issue the following code.

```
Source timesSel = timesDim.selectValue("1996");
```

Example: Selecting based on key values and element values

Assume that you have three primary `Source` objects named `productsDim`, `promotionsDim`, `channelsDim`, and `timesDim`, that you got from `MdmDimension` objects and that you have a primary `Source` object named `sales` that you got from an `MdmMeasure` object. The `productsDim`, `promotionsDim`, `channelsDim`, and `timesDim` objects do not have any keys (that is, they are not dimensional). The sales Source object is multidimensional. It has `productsDim`, `promotionsDim`, `channelsDim`, and `timesDim` as dimensions or keys.

To select all of the products that sold more than $10,000,000 in 1996, you can issue the following code.

```
Source promotionSel = promotionsDim.selectValue("Promo total");
Source channelSel = channelsDim.selectValue("Channel total");
Source timeSel = timesDim.selectValue("1996");
Source bigSellers = productsDim.select(sales.gt(10000000)).
    join(promotionSel).join(timeSel).join(channelSel);
```

Selecting Elements Based on Rank

What is ranking?

When a `Source` is sorted according to some attribute (or attributes), then the position of the elements of the `Source` represents a kind of ranking — the so-called unique ranking. Finding the position of an element is discussed in “Finding the position of elements” on page 6-7.

There are many other types of rankings that are not unique and that are called variant rankings. These are discussed in “Ranking elements in different ways” on page 6-10.

Finding the position of elements

The `position` method returns a `Source` that represents the position of any given element in the original `Source`. The new `Source` has the type of Integer and has the original `Source` as an input. The `position` method returns a `Source` that represents the position of any given element of the base `Source`. If the base `Source` is sorted according to some attribute (or attributes), then the position represents a kind of ranking - the so called unique ranking.

You can also use the shortcut methods described in the following table to find elements based on their position in a `Source` object or to find the position of elements with the specified value or values.

Method	Description
`at(pos)`	Identifies the values of elements in a `Source` that have the specified position. There are two versions of this method. One version allows you specify the position using a `Source` object; in the other, you specify position using an `int` value.
`first()`	Identifies the element or elements in a `Source` that have position 1.
`last()`	Identifies the element or elements in a `Source` that have the largest position value.
`positionOfValue(value)`	Identifies the positions of elements in a `Source` that have the specified value. There are two versions of this method. One version allows you specify the value using a `Source`; in the other, you specify value using a `String`.
`positionOfValues(values)`	Identifies the positions of elements in a `Source` that have the specified values. There are two versions of this method. One version allows you specify the value using a `Source`; in the other, you specify value using an array of `String` objects.

Example: Finding the positions of elements when there are no keys

Assume that there is a `Source` named `products` whose elements are the unique identifiers of products. To create a new `Source` whose elements are the positions of the elements of `products`, issue the following code.

```
Source productsPosition = products.position();
```

A tabular representation of `productsPosition` showing the position of the elements in `products` is shown below. Note that the `position()` method is one based.

Input	Element
`products`	**Integer**
395	1
400	2
405	3
415	4
420	5
425	6
...	

Example: Finding the positions of elements when there are inputs

Assume that there is a `Source` named `unitsSoldByCountry` (shown below) that has an input of `products`, an output of `countries`, and elements whose values are the total number of units for each product sold for each country.

Input	Output	Element	Position
`product`	**`countries`**	**Integer**	
395	Australia	500	1
395	United States	800	2
...	...		...
49780	Australia	10000	1
49780	United States	50	2
49780	...		...

To create a new `Source` named `positionUnitsSoldByCountry` whose elements are the positions of the elements of `unitsSoldByCountry`, issue the following code.

```
Source positionUnitsSoldByCountry = unitsSoldByCountry.position();
```

Ranking elements in different ways

The following table provides example code for ranking elements in different ways where the `Source` (named `base`) whose elements you want to rank has two inputs named `input1` and `input2`.

Rank	Example Code
ascending	`Source sortedTuples = base.sortAscending()`
descending	`Source sortedTuples = base.sortDescending()`

same order as another `Source`	`Source sortedTuples = base.sortAscending`
	`   (Source sortValue)`

reverse order as another `Source`	`Source sortedTuples = base.sortDescending`
	`   (Source sortValue)`

minimum	`Source sortedTuples = base.join(input1).sortDescending(input2);`
	`Source equivalentRankedTuples =`
	`  sortedTuples.join(input2, input2);`
	`NumberSource minRank = sortedTuples.`
	`  positionOfValues(equivalentRankedTuples).minimum();`
	
	The minimum ranking differs from unique ranking in the way it deals with ties (elements in the `Source` that share the same value for the attribute). All ties are given the same rank, which is the minimum possible.

maximum	`Source sortedTuples = base.join(input1).sortDescending(input2);`
	`Source equivalentRankedTuples =`
	`   sortedTuples.join(input2, input2);`
	`NumberSource maxRank = sortedTuples.positionOfValues`
	`     (equivalentRankedTuples).maximum();`
	
	The maximum ranking differs from unique ranking in the way it deals with ties (elements in the `Source` that share the same value for the attribute). All ties are given the same rank, which is the maximum possible rank.

Rank	Example Code

average	`Source sortedTuples = base.join(input1).sortDescending(input2;`
	`Source equivalentRankedTuples =`
	`  sortedTuples.join(input2, input2);`
	`NumberSource averageRank = sortedTuples.positionOfValues`
	`   (equivalentRankedTuples).average();`
	
	The average ranking differs from unique ranking in the way it deals with ties (elements in the `Source` that share the same value for the attribute). All ties are given the same rank, which is equal to the average unique rank for the tied values.

packed	`Source tuples = base.join(o`
	`   utput1);`
	`Source firstEquivalentTuple = tuples.join(input2, input2.first();`
	`Source packedRank = firstEquivalentTuple.join(tuples).`
	`  sortDescending(input2).positionOfValues(base.value().`
	`  join(time.value());`
	
	Packed ranking, also called dense ranking, is distinguished from minimum ranking by the fact that the ranks are packed into consecutive integers.

percentile	`Source sortedTuples = base.join(input1).sortDescending(input2);`
	`Source equivalentRankedTuples =`
	`  sortedTuples.join(input2, input2);`
	`NumberSource minRank = sortedTuples.`
	`     positionOfValues(equivalentRankedTuples).minimum();`
	`NumberSource percentile = minRank.minus(1).times(100).`
	`     div(sortedTuples.count());`
	
	This code uses the following formula to calculate the percentile of an attribute `A` for a `Source` S with `N` elements.
	
	`Percentile(x) =  number of elements`
	`(for which the A differs from A(x))`
	`that come before x in the ordering * 100 / N`
	
	The percentile, then, is equivalent to the `minimum rank -1 * 100 / N`.

Rank	Example Code

ntile	`NumberSource n = ...;`
	`Source sortedTuples = base.join(input1).sortDescending(input2);`
	`NumberSource uniqueRank = sortedTuple.`
	`      positionOfValues(base.value().join(input1.value());`
	`NumberSource ntile = uniqueRank.times(n).`
	`     div(sortedTuples.count()).ceiling();`
	
	In this code, the ntile ranking for a given *n* is defined by dividing the ordered `Source` of size count into *n* buckets, where the bucket with rank `k` is of size. The ntile rank is equivalent to the following formula.
	
	`ceiling*((uniqueRank*n)/count)`.

Selecting Elements Based on Hierarchical Position

Primary Source objects that you use to navigate a hierarchy

To navigate with a hierarchy you need to create two primary Source objects: a primary `Source` that corresponds to the hierarchy, and a primary `Source` that represents the parent-child relationships within this hierarchy.

Creating a primary Source that represents a default hierarchy

To do create a primary `Source` that represents a default hierarchy, you take the following steps:

1. Retrieve the default hierarchy of the `MdmDimension` by taking the following steps:

   a. Check to see if the `MdmDimension` is a union dimension by checking to see if it has an `MdmUnionDimensionDefinition`.

   b. If the `MdmDimension` has an `MdmUnionDimensionDefinition`, then check to see if it has a regions that are `MdmHierarchy` objects.

   c. If the `MdmDimension` has regions that are `MdmHierarchy` objects, select the `MdmHierarchy` that is its default hierarchy.

2. Make the default hierarchy a `Source` object, by calling the `getSource` method on it.

Sample code: Retrieving a default hierarchy

The `getMyDefaultHierarchy` retrieves the default hierarchy of an `MdmDimension` is shown below. This method calls the `getMyRegions` method that retrieves the regions of an `MdmDimension` which, in turn, calls the `getMyMdmUnionDimensionDefinition` method that checks to see if the `MdmDimension` is a union dimension.

```
// method that gets all of the Regions of an MdmDimension
private MdmHierarchy getMyDefaultHierarchy(MdmDimension mdmDim) {
      List hierarchies = getMyRegions(mdmDim);
      if ( hierarchies == null )
        return null;
      for (Iterator iterator = hierarchies.iterator(); iterator.hasNext();) {
        MdmHierarchy hier = (MdmHierarchy) iterator.next();
        if (hier.hasMdmTag(MdmMetadataProvider.DEFAULT_HIERARCHY_TAG))
          return hier;
      }
    return null;
  }

// method that gets all of the Regions of an MdmDimension
private List getMyRegions(MdmDimension mdmDimension ) {
      MdmUnionDimensionDefinition unionDimDef =
   getMyMdmUnionDimensionDefinition ( mdmDimension );
      if ( unionDimDef != null )
        return unionDimDef.getMyRegions();
    return null;
  }

// method that checks to see if MdmDimension is a UnionDimension
private MdmUnionDimensionDefinition getMyMdmUnionDimensionDefinition(
MdmDimension
      mdmDimension ) {
      MdmDimensionDefinition dimDef = mdmDimension.getDefinition();
      if((dimDef == null) || (!(dimDef instanceof MdmUnionDimensionDefinition)))
        return null;
      return (MdmUnionDimensionDefinition) dimDef;
    return null;
  }
```

Creating a primary Source for the parent-child relationship

If an `MdmHierarchy` is a level hierarchy, it's elements are in parent-child relationship to each other. To create a `Source` object that represents the parent-child relationships within a hierarchy, you take the following steps:

1. Create an `MdmAttribute` that represents the parent-child relationships by using the `getParentRelation` method on the `MdmHierarchy`.
2. Create a `Source` from the MdmAttribute created in step 1 by using the `getSource` method.

Creating Source objects for other relationships

A feature of the OLAP API representation of a relation, such as a parent-child relation, is that it is directional. A `Source` object that represents a parent-child relation maps the children to the parent, but not the parents to the children. By contrast, in SQL a table that represent thea realtionship is non-directional. The basic reason is that the OLAP API, unlike SQL, uses the structure of `Source` objects to automatically determine how they `join`. Since in the OLAP API relations are directional, if you want a relation to be in the opposite direction, you need to invert it.

Assume that there is a `Source` named `parentChild` on a hierarchy named `levelHierarchy`. To create `Source` objects that represent other relationships, you `join` these two `Source` objects in different ways. In other words, as shown in the followng table, you can create new `Source` objects that represent the children, siblings, and grandparents in the hierarchy by using the `join` method on the `Source` that represents the parentCihld relation.

Code	Description
`Source childParent = levelHierarchy.join(parentChild, levelHierarchy.value());`	Selects those elements of the hierarchy whose parent is the given element.
`Source siblingParent = levelHierarchy.join(parentChild, parent);`	Selects those elements of the hierarchy whose parent is the same as the parent of the given element.
`Source grandParent = parentChild.join(levelHierarchy, parentChild);`	Selects those elements of the hierarchy whose parents are the parents of the given element.

Example: Drilling down

Assume that there is an MdmDimension object for which you have created a Source named productsDim. Assume also that this MdmDimension object has a default hierarchy for which you have created an MdMHierarchy called prodStdHierObj and a Source called prodHeir. You use the following code to drill down the "Trousers - Women" division of the hierarchy.

```
// Get the parent relation from the hierarchy
MdmAttribute prodHierParentObj = prodStdHierObj.getParentRelation();
StringSource prodHierParent = prodHierParentObj.getSource();
// Select children of Trousers - Women
// - Reverse the parent relation to get a children relation
Source prodHierChildren = prodHier.join(prodHierParent, prodHier.value());
// - Note the join is hidden because we only want the children of
// - Trousers - Women, and not Trousers - Women itself
Source trousersChildren = prodHierChildren.join(prodHier,
   context.getDataProvider().createConstantSource("Trousers - Women"), false);
// Select Shirts - Boys, Trousers - Women, and Shorts - Men
Source prodHierSel = prodHier.selectValues(new String[]
  {"Shirts - Boys","Trousers - Women","Shorts - Men"});
// Insert the children of Trousers - Women after Trousers - Women
// (which is 2nd value)
Source drilledProdHierSel = prodHierSel.appendValues(trousersChildren);
// This selection has the effect of sorting the result in hierarchical order.
Source result = prodHier.selectValues(drilledProdHierSel);
```

Creating a Source with duplicate inputs

Suppose we want to do a region-to-region comparison in some way. Specifically, suppose we want to create a data view in which the regions appear on both the rows and the columns. In the OLAP API you use the alias() and the value() methods to do this. The alias() method creates a new Source that mirrors exactly the original Source in terms of its data, its inputs, and its outputs. The only difference is that the original Source becomes the type of the alias Source. The value() method creates a new Source that has the original Source as both its type and as an input.

Technique for creating duplicate inputs

Assume that there would naturally be an input-output match between input A of the original Source (called base) and some output B of the joined Source in the join shown below.

```
Source result = base.join(joined, comparison);
```

To avoid this input-output match, and hence keep A as an input of the result, use the following procedure.

```
//Create an alias for B called B2;
Source B2 = B.alias();
//Create a variant of the original called base2
//We know that input A will match to B
Source base2 = base.join(B, B2.value());
//Now join base2 and joined
//We know that input B2 will not match to B in joined
Source preResult = base2.join(joined, comparison);
//Finally, join to the B2 and regain the input A
Source result = preResult.join(B2, A.value());
```

Example: Creating a Source with duplicate inputs or outputs

Assume that we have a `Source` named `region` that does not have any inputs or outputs and whose elements are the names of geographical regions. Assume also that we want to create a data view in which the regions appear on both the rows and the columns. For each cell in this table we want to show the percentage difference between the areas (in square miles) of the regions. In other words, we want to create a `Source` named `regionComparison` that has two inputs -- both of them the `Source` named `regions`.

The following code shows how you do this.

```
//Create an alias for region that is for the row
Source rowRegion = region.alias();
//Create an alias for region that is for the column
Source columnRegion = region.alias();
//Create rowRegionArea which has an input of rowRegion,
//     an output of area,
//     and elements whose values are the same as those of region
Source rowRegionArea = area.join(rowRegion.value());
//Create columnRegionArea which has an input of columnRegion,
//     an output of area,
//     and elements whose values are the same as those of region
Source columnRegionArea = area.join(columnRegion.value());
//Compute the values of the cells
Source areaComparison = rowRegionArea.div(columnRegionArea).times(100);
//Create a new Source with outputs rather than inputs
Source regionComparison = areaComparison.join(rowRegion.join(columnRegion))
```

The first two lines of code create two new `Source` objects that are aliases for the `Source` named `region`. These `Source` objects are called `rowRegion` and `columnRegion`.

The next two lines of code create `Source` objects, named `rowRegionArea` and `columnRegionArea`, that represent the areas of `rowRegion` and `columnRegion` respectively. To create `rowRegionArea`, we `join` `area` which has the input of `region` to `rowRegion.value()` which has an input of `rowRegion` and the same elements as `region`. The `rowRegionArea` `Source` has an input of `rowRegion`, an output of `area`, and elements whose values are the same as those of `region`. To create `columnRegionArea`, we `join` area which has the input of `region` to `columnRegion.value()` which has an input of `columnRegion` and the same elements as `region`. The `Source` named `columnRegionArea` has an input of `columnRegion`, an output of `area`, and elements whose values are the same as those of `region`. These `join` calls have the effect of replacing the `region` input with `rowRegion` or `columnRegion`, which, since they both have the names as regions as data, makes no real difference to the value of `area`.

The next line of code performs the needed computation. Because `rowRegionArea` has `rowRegion` as an input and `columnRegionArea` has `columnRegion` as an area, the new `Source` named `areaComparison` has two inputs, `rowRegion` and `columnRegion`, both of whose elements are the names of regions. What we have done is to effectively create a `Source` object that has duplicate inputs.

The final step of changing inputs to outputs is easy. We merely `join` `areaComparison` to its inputs (`rowRegion` and `columnRegion`).

7

Performing Calculations

Chapter summary

This chapter discusses how you perform calculations using the OLAP API.

List of topics

This chapter includes the following topics:

- Performing Numerical Operations
- Making Numerical Comparisons
- Working with Standard Numerical Functions
- Working with Aggregation Methods
- Creating Your own Numerical Functions
- Working With Strings

Performing Numerical Operations

How do you perform numerical operations?

Using the OLAP API you perform basic numeric operations using `NumberSource` methods such as `minus`. There are separate versions of each of these methods that you can use to specify a literal `double`, `float`, `int`, or `short` value. There is also a version of each of these method that takes a `NumberSource` as an argument.

The OLAP API methods that you use to perform basic numeric operations include those outlined in the following table.

Method	Description
`div(rhs)`	Divides the value of each element of the `NumberSource` by the specified value.
`intpart()`	Identifies the integer portion of the value of each element of the `NumberSource`.
`minus(rhs)`	Subtracts the specified value from the value of each element of the `NumberSource`.
`negate()`	Negates the value of each of the elements of the `NumberSource`.
`plus(rhs)`	Adds the specified value to the value of each element of the `NumberSource`
`rem(rhs)`	Divides the value of each element of the `NumberSource` by the specified double value and determines the remainder for each operation.
`times (rhs)`	Multiplies the value of each element of the `NumberSource` by the specified value.

Example: Subtracting the same value from all elements

Assume, as shown below. that there is a `NumberSource` named `unit_Cost` that has outputs of `productsDim` and `timesDim` and a type of Integer.

Output2	Output1	Element
`productsDim`	**`timesDim`**	**`unit_Cost`**
Boys	1998	4000
Boys	...	...
Boys	31-DEC-01	10
...	...	...
49780	1998	500
49780	...	...
49780	31-DEC-01	9

Now assume that you want to subtract 10% of the sales from each element to find the adjusted income for each product. To do this you use the following code.

```
NumberSource percentAdjustment = unit_Cost.minus(unit_Cost.times(.10));
```

The new `NumberSource`, named `percentAdjustment`, has the following structure and values.

Output2	Output1	Element
`productsDim`	**`timesDim`**	**`unit_Cost`**
Boys	1998	3600
Boys	...	...
Boys	31-DEC-01	9
...	...	...
49780	1998	450
49780	...	...
49780	31-DEC-01	8

Example: Subtracting the values of one NumberSource from another

Assume that you have the `NumberSource` named `unitCost` described in the previous example and that you also have the `NumberSource` named `unitManufacturingCost` shown below.

Output	Output	Element
`productsDim`	**`timesDim`**	**Integer**
Boys	1998	600
Boys	...	...
Boys	31-DEC-01	3
...	...	...
49780	1998	250
49780	...	...
49780	31-DEC-01	2

Now assume that you want to calculate the non-manufacturing for each product. To do this you need to subtract the manufacturing costs from the unit costs using the code shown below.

```
NumberSource nonManufacturingCost = unitCost.minus(unitManufacturingCost);
```

The new `NumberSource` has the structure and values shown below.

Output	Output	Element
`productsDim`	**`timesDim`**	**Integer**
Boys	1998	3400
Boys	...	...
Boys	31-DEC-01	7
...	...	...
49780	1998	250
49780	...	...
49780	31-DEC-01	7

For a more complete explanation of these methods, see *Oracle OLAP API Reference*.

Making Numerical Comparisons

How do you make numerical comparisons?

The `NumberSource` class has a number of methods make numerical comparisons. These methods compare the value of each element in a `NumberSource` to a specified value. These methods return a `BooleanSource` that has the same structure as the original `NumberSource` and that has an element that is true when the comparison for a given element of the original `NumberSource` is true, or false when the comparison is false. There are separate versions of each of these methods that you can use to specify a literal `double`, `float`, `int`, or `short` value.

List of numerical comparison methods

The numerical comparison methods provided with the OLAP API include those listed in the following table.

Method	Description
`eq`	Compares each element of the `NumberSource` to the specified value, and determines if it is an equal value
`ge`	Compares each element of the `NumberSource` to the specified value, and determines if it is a greater or equal value.
`gt`	Compares each element of the `NumberSource` to the specified value, and determines if it is a greater value.
`le`	Compares each element of the `NumberSource` to the specified value, and determines if it is a lesser or equal value.
`lt`	Compares each element of the `NumberSource` to the specified value, and determines if it is a lesser value.
`ne`	Compares each element of the `NumberSource` to the specified value, and determines if it is an unequal value.

For a more complete explanation of these methods, see *Oracle OLAP API Reference*.

Working with Standard Numerical Functions

How do the standard numerical functions work?

The OLAP API has many methods that represent standard numerical functions. When you use these functions with a `NumberSource`, they return a new `NumberSource` that has the same structure as the original `NumberSource` and whose elements have the values of the original `NumberSource` modified according to the function. For example, the `abs()` method returns a new `NumberSource` each of whose elements has the absolute value of the value of the corresponding element in the original `NumberSource`.

You can also write your own functions as described in “Creating Your own Numerical Functions” on page 7-10.

List of methods that represent standard functions

The OLAP API methods that represent standard functions include those listed in the following table.

Method	Description
`abs()`	Calculates the absolute value of each element of the `NumberSource`.
`arccos()`	Calculates the angle value (in radians) of the value (interpreted as a cosine) of each element of the `NumberSource`.
`arcsin()`	Calculates the angle value (in radians) of the value (interpreted as a sine) of each element of the `NumberSource`.
`arctan()`	Calculates the angle value (in radians) of the value (interpreted as a tangent) of each element of the `NumberSource`.
`cos()`	Calculates the cosine of the value (interpreted as an angle value in radians) of each element of the `NumberSource`.
`cosh()`	Calculates the hyperbolic cosine of the value (interpreted as an angle value in radians) of each element of the `NumberSource`
`log()`	Calculates the natural logarithm of the value of each element of the `NumberSource`.
`pow(rhs)`	Raises the value of each element of the `NumberSource` to the specified value.
`round(multiple)`	Rounds the value of each element of the `NumberSource` to the nearest multiple of the specified value.
`sin()`	Calculates the sine of the value (interpreted as an angle) of each element of the `NumberSource`.
`sinh()`	Calculates the hyperbolic sine of the value (interpreted as an angle) of each element of the `NumberSource`.
`sqrt()`	Calculates the square root of each element of the `NumberSource`.
`tan()`	Calculates the tangent of the value (interpreted as an angle) of each element of the `NumberSource`.
`tanh()`	Calculates the hyperbolic tangent of the value (interpreted as an angle) of each element of the `NumberSource`.

For a more complete explanation of these methods, see Oracle OLAP API Reference.

Working with Aggregation Methods

What are the aggregation methods?

The numerical aggregation methods provided by the OLAP API include the methods in the following table. You can also write your own aggregation functions as described in "Creating Your own Numerical Functions" on page 7-10.

Method	Description
`average`	Calculates the average of the values of a `NumberSource`.
`maximum`	Identifies the largest value of a `NumberSource`.
`minimum`	Identifies the smallest value of a `NumberSource`.
`total`	Calculates the sum of the values of a `NumberSource`.

There are two different versions of each of the numerical aggregation methods. One version excludes all null values when making its calculations. The other version allows you to specify whether or not you want null values included in the calculation. Each version returns a new `NumberSource` that, for each set of input values, has an element whose value is the sum of all of the elements in the original `NumberSource` that have the same set of input values

How do the aggregations methods work?

Standard numerical methods like `stdev()` work on each element in a `NumberSource`. An aggregation method is a method like `total()` that uses the values in a series of `Source` elements to perform its calculations. When a `NumberSource` does not have any inputs, this method creates a new `NumberSource` with a single element whose value is the sum of the values of the elements in the original `NumberSource`.

When a `NumberSource` has inputs, each set of input values identifies a subset of elements (tuples) that are arranged by the outputs of the `NumberSource` (if any). In this case, an aggregation method works on each set of uniquely positioned elements. In other words, when a `NumberSource` has inputs, an aggregation method calculates the result of the function for each subset.

For more information on how OLAP API methods determine the position of an element and therefore how they determine what elements to use when calculating the values of aggregation methods, see "Finding the position of elements" on page 6-7.

Example: Calculating the sum of the elements when a Source does not have inputs

Assume that you have the `Source` named `unitsSoldByCountry` (shown below) whose elements are the total number of units for each product sold for each country.

Output	Output	Element
`products`	**`countries`**	**Integer**
395	Australia	1300
395	United States	800
...	...	
49780	Australia	10050
49780	United States	50
49780	...	

Now assume that you want to total these values. Since both `products` and `countries` are outputs, when you issue the code shown below, the new `NumberSource` calculates the total number of units sold for all products in all countries.

```
NumberSource totalUnitsSold = unitsSoldyByCountry.total();
```

The new `NumberSource` called `totalUnitsSold` has only a single element that is the total of the values of the elements of `unitsSoldByCountry`.

Element
Integer
11350

Example: Calculating the sum of the elements when a Source has inputs

Assume that you have the `Source` named `unitsSoldByCountry` (shown below) whose elements are the total number of units for each product sold for each country.

Output	**Input**	**Element**
countries	**products**	**Integer**
Australia	395	1300
Australia	49780	10050
...	...	...
United States	49780	50
United States	395	800
...	...	

Now assume that you total these values. Since `product` is input, when you issue the code shown below, the new `NumberSource` calculates the total number of units sold for all products in each countries;. It does not calculate the total for all products in all countries.

```
NumberSource totalUnitsSoldByCountry = unitsSoldByCountry.total();
```

The new `NumberSource` called `totalUnitsSoldByCountry` has the structure and values shown below.

Input	**Elements**
countries	**Integer**
Australia	11350
...	...
United States	850
...	...

Creating Your own Numerical Functions

Creating parameters

The alias method can be used to create parameters. "Example: Creating a function" on page 7-10 shows how to create a new function using the `alias` method. You can only create cell or row calculation functions in this way. To create client aggregation or position-based functions you use the `extract` method.

Example: Creating a function

The following function takes a number and multiplies it by 1.05. The function has one parameter, called `param`, which is created by calling the `alias` method on the fundamental `Source` representing the Number OLAP API data type which is the set of all numbers. Note how the `value` method is used to make the parameter an input of the function.

```
//Get the Source that represents the number data type
NumberSource number =(NumberSource)dataProvider
.getFundamentalDefinitionProvider()
.getNumberDataType()
.getSource();
//Create a parameter
NumberSource param = (NumberSource)number.alias();
//Create a function
NumberSource function = ((NumberSource)param.value()).times(1.05);
```

The function created in this way is effectively the same as the built-in functions provided by the OLAP API. It can be used by joining the function to the parameter and the required parameter expression as shown below. You can then apply the function to a `Source` named `sales` as shown below.

```
//Use the function
NumberSource sales = ...;
NumberSource fsales = function.join(param, sales);
```

Example: Creating a parameterized selection

Assume you want to create a product selection defined to be the set of all products for which the `unitsSold` measure is greater than the value specified by a parameter. The parameter must be specified before data can be fetched from this `Source`. You can create this parameter using the following code.

```
//Get the Source that represents the number data type
```

```
NumberSource number = dataProvider
.getFundamentalDefinitionProvider()
.getNumberDataType()
.getSource();
//Create a parameter
NumberSource param = (NumberSource)number.alias();
//Create a parameterized selection
Source products = ...;
NumberSource unitsSold = ...;
Source productSelection = products.select(unitsSold.gt(param.value()));
```

To set the value of the parameter to 100, you write the following code.

```
Source unitsSoldGT100 = productSelection.join(param, 100);
```

Example: Creating an aggregation function

Assume that you want to create a weighted average function. To do so, you write the following code.

```
//Define an aggregation function
NumberSource weight = ...;
//Create a parameter
NumberSource param = (NumberSource) number.alias();
//Create a function
NumberSource weightedAverage = param.extract().times(weight).average();
```

As with the example of a standard function “Example: Creating a function” on page 7-10, this code first creates a parameter named `param` for the function to use. However, since this is an aggregation function, the code uses the `extract()` method with `param` when it calculates the final result.

To use this function, you issue the following code.

```
//Use the aggregation function
NumberSource sales = ...;
NumberSource paramSales = dp.createConstantSource(param.selectValues(sales));
Source weightedSales = weightedAverage.join(paramSales);
```

Working With Strings

String manipulation methods

The `StringSource` class defines methods that are string-specific versions of various `Source` methods that you can use to append, insert, select, and remove elements whose values are Java `String` objects. The `StringSource` class also has methods that you can use to manipulate the elements of the `StringSource` objects. These methods include those listed in the following table.

Method	Description
`length()`	Determines the length of each element of the `StringSource`.
`textFill(width)`	Reformats each element of the `StringSource` to the specified width by adding blank spaces.
`toLowercase()`	Converts the alphabetic characters of each element of the `StringSource` into lowercase.
`toUppercase()`	Converts the alphabetic characters of each element of the `StringSource` into uppercase.
`trim()`	Removes the leading and trailing blank spaces from each element of the `StringSource`.
`trimLeading()`	Removes the leading blank spaces from each element of the `StringSource`.
`trimTrailing`	Removes the trailing blank spaces from each element of the `StringSource`.

Substring methods

The OLAP API provides the methods that you can use to manipulate substrings within the elements of a `StringSource`. These methods include those listed in the following table.

Method	Description
`indexOf (substring, fromIndex)`	Searches each element of the `StringSource` beginning at the specified character position and identifies the position of the first character of the specified substring.
`remove (index, length)`	Removes the characters between the specified character positions from each element of the `StringSource`
`replace (oldString, newString)`	Searches each element of the `StringSource` for the specified substring, and replaces it with a different substring when it is found
`substring (index, length)`	Selects the characters between the specified character positions from each element of the `StringSource`

There are two different versions of each of these methods. In one version you specify the values using `Source` objects, in the other you specify the values using literal values.

8 Using a TransactionProvider

Chapter summary

This chapter is 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 on the `TransactionProvider` to prepare and commit a `Transaction` before you can create a `Cursor` for a derived `Source`.

List of topics

This chapter includes the following topics:

About Transaction Objects

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 on another `Source`, that `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 on a `TransactionProvider`. In a child `Transaction` you can change 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 performing what-if analysis.

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 an OLAP service. For more information on `Cursor` objects, see Chapter 9.
- A *write* `Transaction`. A write `Transaction` is required for creating a derived `Source` or for changing the state of a `Template`. For more information on creating a derived `Source`, see Chapter 5. For information on `Template` objects, see Chapter 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.

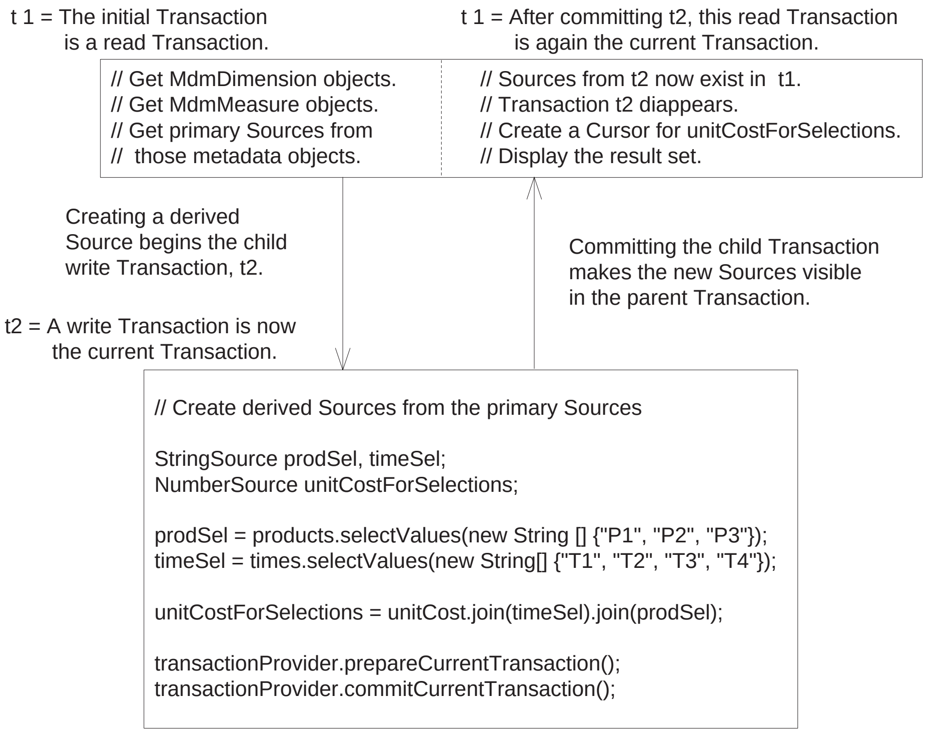
Before you can create a `Cursor` to fetch the result set specified by a derived `Source`, you must move the `Source` from the child write `Transaction` into the parent read `Transaction`. To do so, you prepare and commit the `Transaction`.

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 on 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`. The figure has a box that represents a read `Transaction`, `t1`, in which an application gets `MdmDimension` and `MdmMeasure` objects and gets primary `Source` objects from them. The application then creates derived `Source` objects by making selections on the primary `Source` objects from `t1`.

When the application calls the `selectValues` method on `products`, a primary `Source`, the write `Transaction` `t2` is created, which is represented by a box below `t1`. The derived `Source` objects `prodSel` and `timeSel` are created in `t2`. Also in `t2` the application joins the derived `Source` objects to the `unitCost` primary `Source` to create the `unitCostForSelections` derived `Source`. The application then calls the `prepareCurrentTransaction` and `commitCurrentTransaction` methods on the `TransactionProvider`, which prepares the `t2` `Transaction` and commits it. Committing `t2` makes the new `Source` objects that were created in `t2` visible and active in `t1`, the parent read `Transaction`. The application can then create a `Cursor` for `unitCostForSelections`. The `t2 Transaction` is no longer active and it disappears.


t 1 = The initial Transaction
is a read Transaction.
t 1 = After committing t2, this read Transaction
is again the current Transaction.
Creating a derived
Source begins the child
write Transaction, t2.
Committing the child Transaction
makes the new Sources visible
in the parent Transaction.
t2 = A write Transaction is now
the current Transaction.

```
// Get MdmDimension objects.
// Get MdmMeasure objects.
// Get primary Sources from
//  those metadata objects.
```

```
// Sources from t2 now exist in  t1.
// Transaction t2 diappears.
// Create a Cursor for unitCostForSelections.
// Display the result set.
```

```
// Create derived Sources from the primary Sources

StringSource prodSel, timeSel;
NumberSource unitCostForSelections;

prodSel = products.selectValues(new String [] {"P1", "P2", "P3"});
timeSel = times.selectValues(new String[] {"T1", "T2", "T3", "T4"});

unitCostForSelections = unitCost.join(timeSel).join(prodSel);

transactionProvider.prepareCurrentTransaction();
transactionProvider.commitCurrentTransaction();
```

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. This creating of alternatives based on an initial state is known as what-if analysis.

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 Transaction, call the beginSubtransaction method on the TransactionProvider you are using. Initially, the child Transaction is a read Transaction. If you then change the state of a Template, 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 Transaction. In the figure, a box represents t1, which is a read Transaction in which a TopBottomTemplate object exists. TopBottomTemplate is an example of a Template that is described in Chapter 11.

A second box represents t2, which is a write Transaction begun when the state of the Template changes to specify selecting the top ten values, which results from the following operations.

```
topNBottom.setTopBottomType(TOP);
topNBottom.setN(10);
```

In t2, the prepareCurrentTransaction and commitCurrentTransaction methods are called on the TransactionProvider. The changes to the state of the TopBottomTemplate become active in t1 and t2 disappears. The getSource method on the DynamicDefinition created by the TopBottomTemplate is called. A Cursor for the Source is created, which retrieves the data from the OLAP service. The values from the Cursor are displayed.

A third box in the figure represents the read Transaction t3, which is a child of t1 that results from the following operation.

```
t1.beginSubtransaction();
```

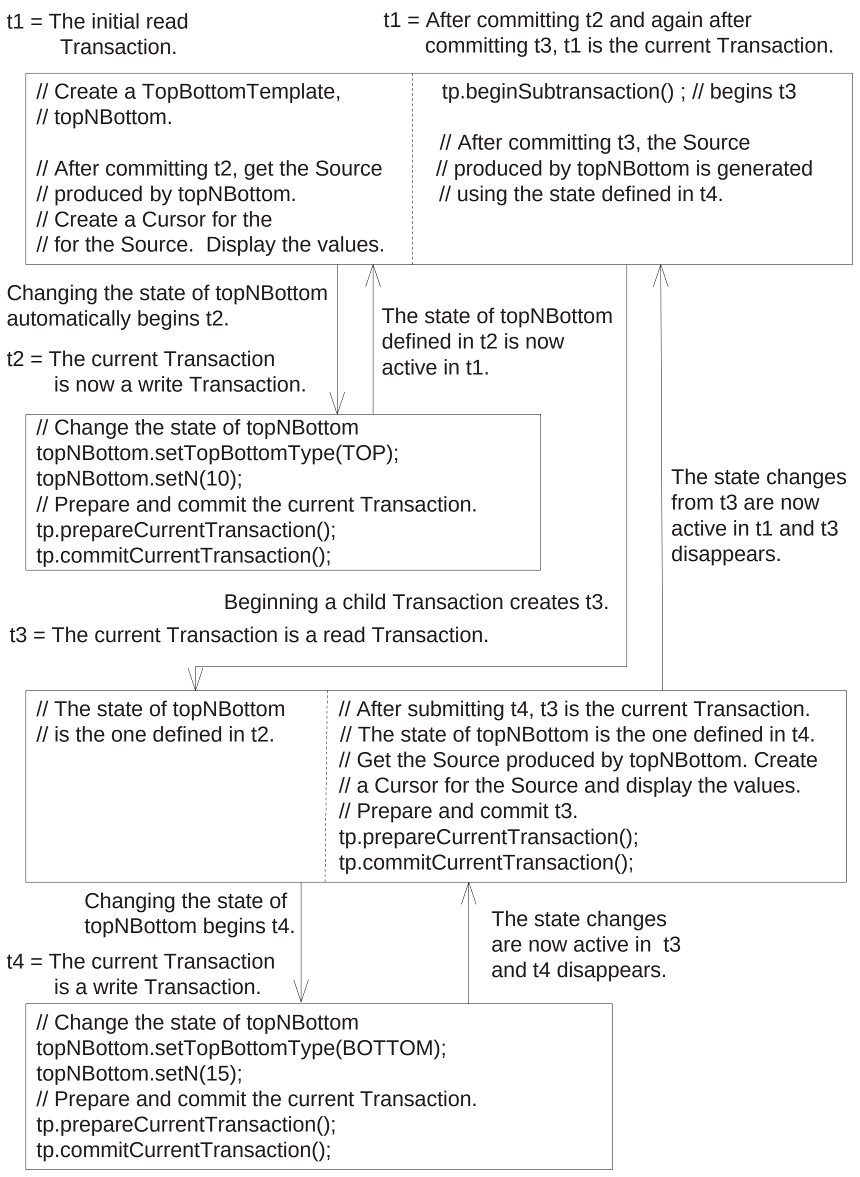
A fourth box represents t4, which is a write Transaction that begins as a result of the following changes to the state of the TopBottomTemplate.

```
topNBottom.setTopBottomType(BOTTOM);
topNBottom.setN(15);
```

The Source produced by the TopBottomTemplate now represents the selection of the bottom fifteen values. In t4, the prepareCurrentTransaction and commitCurrentTransaction methods are called on the TransactionProvider. The changes to the state of the TopBottomTemplate made in t4 become active in t3 and t4 disappears.

In t3, the getSource method is called on the DynamicDefinition, a Cursor for the Source is created, and values from the Cursor are displayed. The prepareCurrentTransaction and commitCurrentTransaction methods are called. The result is that the state of the TopBottomTemplate from t3, representing the bottom fifteen values, moves into t1 and replaces the state of the TopBottomTemplate that represented the top ten values. The t3 Transaction disappears and t1 is again the current Transaction.

In the following figure, `tp` is the `TransactionProvider`.



```
// Create a TopBottomTemplate,
// topNBottom.

// After committing t2, get the Source
// produced by topNBottom.
// Create a Cursor for the
// for the Source.  Display the values.
```

```
tp.beginSubtransaction() ; // begins t3

// After committing t3, the Source
// produced by topNBottom is generated
// using the state defined in t4.
```

```
// Change the state of topNBottom
topNBottom.setTopBottomType(TOP);
topNBottom.setN(10);
// Prepare and commit the current Transaction.
tp.prepareCurrentTransaction();
tp.commitCurrentTransaction();
```

```
// The state of topNBottom
// is the one defined in t2.
```

```
// After submitting t4, t3 is the current Transaction.
// The state of topNBottom is the one defined in t4.
// Get the Source produced by topNBottom. Create
// a Cursor for the Source and display the values.
// Prepare and commit t3.
tp.prepareCurrentTransaction();
tp.commitCurrentTransaction();
```

```
// Change the state of topNBottom
topNBottom.setTopBottomType(BOTTOM);
topNBottom.setN(15);
// Prepare and commit the current Transaction.
tp.prepareCurrentTransaction();
tp.commitCurrentTransaction();
```

t1 = The initial read Transaction.
t1 = After committing t2 and again after committing t3, t1 is the current Transaction.
Changing the state of topNBottom automatically begins t2.
t2 = The current Transaction is now a write Transaction.
The state of topNBottom defined in t2 is now active in t1.
The state changes from t3 are now active in t1 and t3 disappears.
Beginning a child Transaction creates t3.
t3 = The current Transaction is a read Transaction.
Changing the state of topNBottom begins t4.
t4 = The current Transaction is a write Transaction.
The state changes are now active in t3 and t4 disappears.

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: Using child Transaction objects” on page 8-11.

About rolling back a Transaction

You roll back, or undo, a Transaction by calling the rollbackCurrentTransaction method on 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: 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.
tp.prepareCurrentTransaction();
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.
tp.prepareCurrentTransaction();
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 on 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 on the `TransactionProvider`, as in the following example.

```
setCurrentTransaction(t1);
```

About TransactionProvider Objects

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.

A `TransactionProvider` has the following methods:

Method	Return Value
`beginSubtransaction`	A `Transaction` that is a child Transaction of the current `Transaction`.
`commitCurrentTransaction`	Void. This method moves any changes made in the current child Transaction into the parent `Transaction`.
`getCurrentTransaction`	The current `Transaction`. Use this method to save the current `Transaction`. You can then set a different saved `Transaction` as the current `Transaction`.
`prepareCurrentTransaction`	Void. This method prepares the current child `Transaction` to be committed into the parent `Transaction`.
`rollbackCurrentTransaction`	Void. This method discards the current child `Transaction`, in effect rolling back any OLAP API operations an application performed in the context of the child `Transaction`. The parent `Transaction` becomes the current `Transaction`.
`setCurrentTransaction`	Void. This method sets a `Transaction` as the current `Transaction`.

As described in “Preparing and committing a Transaction” on page 8-3, 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, you might also use the other methods on 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.

Example: Using child Transaction objects

To demonstrate how to use Transaction objects to modify dynamic queries, the following example 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.

Replace the last five lines of the code from the TopBottomTest class with the following.

```
/*
 * 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(15);
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{
  context.getTransactionProvider().prepareCurrentTransaction();
}
catch(NotCommittableException e){
 System.out.println("Caught exception " + e + ".");
}
context.getTransactionProvider().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.
 */
createCursor(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{
  context.getTransactionProvider().prepareCurrentTransaction();
}
```

```
catch(NotCommittableException e){
  System.out.println("Caught exception " + e + ".");
}
context.getTransactionProvider().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.
createCursor(topNBottom.getSource());

// Commit the grandchild into the child.
try{
  context.getTransactionProvider().prepareCurrentTransaction();
}
catch(NotCommittableException e){
  System.out.println("Caught exception " + e + ".");
}
context.getTransactionProvider().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{
  context.getTransactionProvider().prepareCurrentTransaction();
}
catch(NotCommittableException e){
  System.out.println("Caught exception " + e + ".");
}
context.getTransactionProvider().commitCurrentTransaction();

/*
 * Transaction parentT1 is now the current Transaction. Again,
 * you could roll back the childT2 Transaction instead of
```

```
     * preparing and committing it. If you did so, then the changes
     * you made in childT2 are discarded. The current Transaction
     * would be parentT1, which would have the original state of
     * the TopBottomTemplate, without any of the changes made in
     * the grandchild or the child transactions.
     */

  }  // end of main() method
}  // end of TopBottomTest class
```

9

Retrieving Query Results

Chapter summary

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 10.

List of topics

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 Fetch Sizes and Fetch Blocks

Retrieving the Results of a Query

Steps in retrieving the results of a query

A query is an OLAP API `Source` that specifies the data that you want to retrieve from an OLAP service and any calculations you want the OLAP service 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 5.
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 on your TransactionProvider. For more information on preparing and committing a Transaction, see Chapter 8.
3. Create a CursorManagerSpecification by calling the createCursorManagerSpecification method on your DataProvider and passing that method the Source.
4. Create a SpecifiedCursorManager by calling the createCursorManager method on your DataProvider and passing that method the CursorManagerSpecification.
5. Create a Cursor by calling the createCursor method on the CursorManager.

Example: Creating a Cursor

The following example 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.

```
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  System.out.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();
CursorManagerSpecification cursorMngrSpec =
                 dp.createCursorManagerSpecification(querySource);
SpecifiedCursorManager cursorMngr =
                           dp.createCursorManager(cursorMngrSpec);
Cursor queryCursor = cursorMngr.createCursor();
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 `Cursor` interface has two subinterfaces: `ValueCursor` and `CompoundCursor`. The Oracle OLAP API has implementations of these subinterfaces. Calling the `createCursor` method on 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. A `ValueCursor` 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: Getting ValueCursor objects from a CompoundCursor” on page 9-5. For an example of a `Source` that has nested output values, see “Example: Getting values from a CompoundCursor with nested outputs” on page 9-6.

Example: Getting a single value from a ValueCursor

An example of a `Source` that represents a single set of values is one returned by the `getSource` method on an `MdmDimension`, such as an `MdmDimension` that represents a hierarchical list of 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.

The following example gets the Source from mdmProductHier, which is an MdmDimension 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.

```
Source productSource = mdmProductHier.getSource();
/*
 * Because productSource 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 productCursor = cursorMngr.createCursor();
// Cast the Cursor to a ValueCursor.
ValueCursor productValues = (ValueCursor) productCursor;
// Set the position to the fifth element of the ValueCursor.
productValues.setPosition(5);
/*
 * Product values are strings. Get the String value at the current
 * position.
 */
Sring value = productValues.getCurrentString();
// Close the SpecifiedCursorManager.
cursorMngr.close();
```

Example: Getting all the values from a ValueCursor

This example uses the same Source as “Getting values from a Cursor” on page 9-3. This example 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.

```
// productValues is the ValueCursor for productSource
productValues.setPosition(1);
```

```
do {
   System.out.println(productValues.getCurrentValue);
}
while(productValues.next());
```

Example: Getting ValueCursor objects from a CompoundCursor

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 on a CursorManager for a Source that represents the values of a measure as specified by selected values from the dimensions of the measure.

The following example uses a Source, named salesAmount, that results from calling the getSource method on an MdmMeasure that represents monetary amounts for sales. The dimensions of the measure are MdmDimension objects representing products, customers, times, channels, and promotions. This example uses Source objects that represent selected values from those dimensions. The names of those Source objects are prodSel, custSel, timeSel, chanSel, and promoSel. The creation of the Source objects representing the measure and the dimension selections is not shown.

The example joins the dimension selections to the measure, which results in a Source named salesForSelections. The example creates a Cursor, named salesForSelCursor, for salesForSelections. The example casts the Cursor to a CompoundCursor, named salesCompndCrsr, 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 dimensions were joined to the measure. In the example, dp is the DataProvider and tp is the TransactionProvider.

```
Source salesForSelections = salesAmount.join(prodSel)
                                       .join(custSel)
                                       .join(timeSel)
                                       .join(chanSel)
                                       .join(promoSel);
// Prepare and commit the current Transaction
try{
  tp.prepareCurrentTransaction();
}
```

```
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesForSelections
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(salesForSelections);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor salesForSelCursor = cursorMngr.createCursor();
// Cast salesForSelCursor to a CompoundCursor
CompoundCursor salesCompndCrsr = (CompoundCursor) salesValues;
// Get the base ValueCursor
ValueCursor specifiedSalesVals = salesCompndCrsr.getValueCursor();
// Get the outputs
List outputs = salesCompndCrsr.getOutputs();
ValueCursor promoSelVals = (ValueCursor) outputs.get(0);
ValueCursor chanSelVals = (ValueCursor) outputs.get(1);
ValueCursor timeSelVals = (ValueCursor) outputs.get(2);
ValueCursor custSelVals = (ValueCursor) outputs.get(3);
ValueCursor prodSelVals = (ValueCursor) outputs.get(4);
/*
 * You can now get the values from the ValueCursor objects.
 * When you have finished using the Cursor objects, close the
 * SpecifiedCursorManager.
 */
cursorMngr.close()
```

Example: Getting values from a CompoundCursor with nested outputs

This example uses the same sales amount measure as “Example: Getting ValueCursor objects from a CompoundCursor” on page 9-5, but it joins the dimension selections to the measure differently. The example 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`, `salesForSelections`, represents a query has nested outputs, which means it has more than one level of outputs.

The `CompoundCursor` that this example creates for `salesForSelections` 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`.

The example joins the selection of promotion dimension values, `promoSel`, to the selection of channel dimension values, `chanSel`. The result is `chanByPromoSel`, a `Source` that has channel values as its base values and promotion values as the values of its output. The example joins to `salesAmount` the selections of product, customer, and time values, and then joins `chanByPromoSel`. The resulting query is represented by `salesForSelections`.

The example prepares and commits the current `Transaction` and creates a `Cursor`, named `salesForSelCursor`, for `salesForSelections`.

The example casts the `Cursor` to a `CompoundCursor`, named `salesCompndCrsr`, and gets the base `ValueCursor` and the outputs from it. In the example, `dp` is the `DataProvider` and `tp` is the `TransactionProvider`.

```
Source chanByPromoSel = chanSel.join(promoSel);
Source salesForSelections = salesAmount.join(prodSel)
                                       .join(custSel)
                                       .join(timeSel)
                                       .join(chanByPromoSel);
// Prepare and commit the current Transaction
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesForSelections
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(salesForSelections);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor salesForSelCursor = cursorMngr.createCursor();
/*
 * Send the Cursor to a method that does different operations
 * depending on whether the Cursor is a CompoundCursor or a
 * ValueCursor.
 */
printCursor(salesForSelCursor);
cursorMngr.close();
/*
 *...Elsewhere in the code is the public printCursor method and the
 * private _printTuple method. 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.
 *
 * 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.
 */

public void printCursor(Cursor rootCursor) {
  int i = 1;
  do {
     output.print(i++ + ": ");
     _printTuple(rootCursor);
     output.print("\n");
     output.flush();
  }
  while(rootCursor.next());
}

private void _printTuple(Cursor cursor) {
  if(cursor instanceof CompoundCursor) {
    CompoundCursor compoundCursor = (CompoundCursor)cursor;
    // Put an open parenthesis before the value of each output
    output.print("(");
    Iterator iterOutputs = compoundCursor.getOutputs().iterator();
    Cursor output = (Cursor)iterOutputs.next();
    _printTuple(output);
    while(iterOutputs.hasNext()) {
      // Put a comma after the value of each output
      output.print(",");
      _printTuple((Cursor)iterOutputs.next());
    }
     // Put a comma after the value of the last output
    output.print(",");
    // Get the base ValueCursor
    _printTuple(compoundCursor.getValueCursor());
    /*
```

```
       * Put a close parenthesis after the base value to indicate
       * the end of the tuple.
       */
      output.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

About Navigating a CompoundCursor

With methods on 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.

Example: Navigating for a table view

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. The following example 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`. It casts the `Cursor` 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 `output` object is a `PrintWriter`.

```
Source unitPriceByDay = unitPrice.join(productSel)
                                 .join(timeSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for unitPriceByDay
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitPriceByDay);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor unitPriceByDayCursor = cursorMngr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;

// Determine a starting position and the number of rows to display
int start = 7;
int numRows = 12;
/*
 * 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 values of the output ValueCursors
   output.print(rootCursor.getOutputs().get(0).getCurrentValue() + "\t");
   output.print(rootCursor.getOutputs().get(1).getCurrentValue() + "\t");
   // Print the value of the base ValueCursor and a new line
   output.print(rootCursor.getValueCursor().getCurrentValue() + "\n");
   output.flush();
};
cursorMngr.close();
```

If the time selection for the query has eight values, such as the first day of each calendar quarter for the years 1999 and 2000, and the product selection has three values, then the result set of the `unitPriceByDay` query has twenty-four positions. The preceding example displays something like the following table, which has the values specified by positions 7 through 18 of the `CompoundCursor`.

```
01-JUL-99    815     57
01-JUL-99    1050    23
01-JUL-99    2055    22
01-OCT-99    815     56
01-OCT-99    1050    24
01-OCT-99    2055    21
01-JAN-00    815     58
01-JAN-00    1050    24
01-JAN-00    2055    24
01-APR-00    815     59
01-APR-00    1050    24
01-APR-00    2055    25
```

Example: Navigating for a crosstab view without pages

This example uses the same query as “Example: Navigating for a table view” on page 9-9. In a crosstab view, the first row is column headings, which are the values from `timeSel` in this example. The output for `timeSel` is the faster varying output because the `timeSel` dimension selection was joined to the measure first. The remaining rows begin with a row heading. The row headings are values from the slower varying output, which is `productSel`. 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 specify the positions of the children of the top-level `CompoundCursor` and then iterate through their positions. The example gets the values but does not include code for putting the values in the appropriate cells of the crosstab display.

```
Source unitPriceByDay = unitPrice.join(productSel)
                                 .join(timeSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();
```

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

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
/*
 * Determine a starting position and the number of rows to display.
 * colStart is the position in columnCursor at which the current
 * display starts and rowStart is the position in rowCursor at
 * which the current display starts.
 */
int colStart = 1;
int rowStart = 1;
String productValue;
String timeValue;
double price;
int numProducts = 3;
int numDays = 12;

// Get the outputs and the ValueCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
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);
ValueCursor unitPriceValues = rootCursor.getValueCursor();// Prices

// Loop through positions of the faster varying output Cursor
for(int pPos = colStart; pPos < colStart + numProducts; pPos++) {
  columnCursor.setPosition(pPos);
  // Loop through positions of the slower varying output Cursor
  for(int tPos = rowStart; tPos < rowStart + numDays; tPos++) {
    rowCursor.setPosition(tPos);
    /*
     * Get the values. Sending the values to the appropriate
     * display mechanism is not shown.
     */
    productValue = columnCursor.getCurrentString();
    timeValue = rowCursor.getCurrentString();
    price = unitPriceValues.getCurrentDouble();
  }
}
cursorMngr.close();
```

The following crosstab view is a display of the values from the result set specified by the `unitPriceByDay` query.

	815	**1050**	**2055**
01-JAN-99	56	22	21
01-APR-99	57	22	21
01-JUL-99	57	23	22
01-OCT-99	56	24	21
01-JAN-00	58	24	24
01-APR-00	59	24	25
01-JUL-00	59	25	25
01-OCT-00	61	25	26

Example: Navigating for a crosstab view with pages

This example creates a `Source` that is based on a sales amount measure. The dimensions of the measure are the customer, product, time, channel, and promotion 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 total sales amount 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 customers, which has three values that specify all of the customers from France, the UK, and the USA. The customer values are the column headings of the crosstab. The next fastest varying output is the selection of products, which has four values that specify types of products. The page dimensions are selections of two time values, which are the first and second calendar quarters of the year 2000, one channel value, which is the direct channel, and one promotion value, which is all promotions.

The `TransactionProvider` is `tp` and the `DataProvider` is `dp`. The `output` object is a `PrintWriter`.

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(salesAmountsForSelections);
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor salesForSelCursor = cursorMngr.createCursor();

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

cursorMngr.close();
// Elsewhere in the code are the private methods.

// This method expects a CompoundCursor.
private void printAsCrosstab(Cursor cursor) {
  // Cast the Cursor to a CompoundCursor
  CompoundCursor rootCursor = (CompoundCursor) cursor;
  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 second 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
        output.println();
        output.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())
      output.println(((ValueCursor) pageIter.next()).getCurrentValue());
    output.println();
  }

  // Prints the column headings on each page
  private void printColumnHeadings(Cursor colCursor) {
    do {
       output.print("\t");
       output.print(((ValueCursor) colCursor).getCurrentValue());
    } while (colCursor.next());
    output.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
      output.print(((ValueCursor) rowCursor).getCurrentValue());
      output.print("\t");
      // Loop over columns
      do {
        // Print data value
        output.print(dataCursor.getCurrentValue());
        output.print("\t");
      } while (colCursor.next());
      output.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);
  }
```

The crosstab output of this example looks like the following.

```
Promotion total
Direct
2000-Q1

                        FR           UK            US
Outer wear - Men        750563.50    938014.00     12773925.50
Outer wear - Women      984461.00    1388755.50    15421979.00
Outer wear - Boys       693382.00    799452.00     9183052.00
Outer wear - Girls      926520.50    977291.50     11854203.00


Promotion total
Direct
2000-Q2

                        FR            UK            US
Outer wear - Men        683521.00     711945.00     9947221.50
Outer wear - Women      840024.50     893587.50     12484221.00
Outer wear - Boys       600382.50     755031.00     8791240.00
Outer wear - Girls      901558.00     909421.50     9975927.00
```

Specifying the Behavior of a Cursor

About 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.
- The *shape* of the *fetch block* of a `Cursor`. The fetch block is the set of elements of each descendent `ValueCursor` that the parent `CompoundCursor` retrieves. The shape of the fetch block is the levels of the `CompoundCursor` at which you set the fetch sizes.

- Whether the OLAP service 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 the OLAP service 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 on the `CursorSpecification` for that `Cursor`. To get the `CursorSpecification` for a `Cursor`, you use methods on 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 10.

Example: Getting CursorSpecification objects from a CursorManagerSpecification

This example 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`.

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

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

```
// 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

About specifying the calculation of extent and the starting and ending positions of a value in its parent

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 the OLAP service to calculate the extent or those positions. To specify the performance of those calculations, you use methods on the CursorSpecification for the Cursor.

Example: Specifying the calculation of the extent of a Cursor

This example specifies calculating the extent of a `Cursor`. The example uses the `CursorManagerSpecification` from "Example: Getting CursorSpecification objects from a CursorManagerSpecification" on page 9-18.

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

You can use methods on 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: Specifying the calculation of starting and ending positions in a parent

This example 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: Getting CursorSpecification objects from a CursorManagerSpecification" on page 9-18.

```
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();
ValueCursorSpecification valCursorSpec = (ValueCursorSpecification)
                                          iterOutputSpecs.next();
while(iterOutputSpecs.hasNext()) {
  valCursorSpec.setParentStartCalculationSpecified(true);
  valCursorSpec.setParentEndCalculationSpecified(true);
  valCursorSpec = (ValueCursorSpecification) iterOutputSpecs.next();
}
```

You can use methods on 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: Calculating the span of the positions in the parent of a value

This example 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 `salesAmountsForSelections` `Source` from “Example: Navigating for a crosstab view with pages” on page 9-13.

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 `output` is a `PrintWriter`.

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

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

 // 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(2); \\ output for time
ValueCursorSpecification prodSelValCSpec =
 (ValueCursorSpecification) outputSpecs.get(3)  \\ output for product

// Specify the calculation of the starting and ending positions
timeSelValCSpec.setParentStartCalculationSpecified(true);
timeSelValCSpec.setParentEndCalculationSpecified(true);
prodSelValCSpec.setParentStartCalculationSpecified(true);
prodSelValCSpec.setParentEndCalculationSpecified(true);

// Create the CursorManager and the Cursor
SpecifiedCursorManager cursorMngr = dp.createCursorManager(cursorMngrSpec);
CompoundCursor cursor = (CompoundCursor) cursorMngr.createCursor();

// Get the child Cursor objects
ValueCursor baseValCursor = cursor.getValueCursor();
List outputs = cursor.getOutputs();
ValueCursor promoSelVals = (ValueCursor) outputs.get(0);
ValueCursor chanSelVals = (ValueCursor) outputs.get(1);
ValueCursor timeSelVals = (ValueCursor) outputs.get(2);
ValueCursor custSelVals = (ValueCursor) outputs.get(3);
ValueCursor prodSelVals = (ValueCursor) outputs.get(4);

// Set the position of the root CompoundCursor
cursor.setPosition(15);
/*
 * Get the values at the current position and determine the span
 * of the values of the time and product outputs.
 */
output.print(promoSelVals.getCurrentValue() + ", ");
output.print(chanSelVals.getCurrentValue() + ", ");
output.print(timeSelVals.getCurrentValue() + ", ");
output.print(custSelVals.getCurrentValue() + ", ");
output.print(prodSelVals.getCurrentValue() + ", ");
output.println(baseValCursor.getCurrentValue());

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

This example produces the following output.

```
Promotion total, Direct, 2000-Q1, Outer wear - Men, US, 9947221.50
The span of Outer wear - Men at the current position is 3.
The span of 2000-Q2 at the current position is 12.
```

Specifying Fetch Sizes and Fetch Blocks

About specifying fetch sizes and fetch blocks

The number of elements of a `Cursor` that the OLAP service sends to the client application during one fetch operation depends on the fetch size specified for that `Cursor`. For a `CompoundCursor`, you can set the fetch size on the `CompoundCursor` itself or at one or more levels of its descendent `Cursor` components. Setting the fetch size on a `CompoundCursor` specifies that fetch size for its child `Cursor` components.

The set of elements the `Cursor` retrieves in a single fetch is the fetch block. The shape of the fetch block is determined by the set of `Cursor` components on which you set the fetch sizes. For more information on fetch sizes and fetch blocks, see Chapter 10.

You specify the shape of the fetch block and the specific fetch sizes according to the needs of your display of the data. To display the results of a query in a table view, you specify the fetch size on the top-level `CompoundCursor`.

To display the results in a crosstab view, you specify the fetch sizes on the children of the top-level `CompoundCursor`. For a crosstab that displays the results of a query that has nested levels of outputs, you might specify fetch sizes at different levels of the children of the component `CompoundCursor` objects.

You use methods on a `CursorSpecification` to set the default fetch size for its `Cursor`. For a `CompoundCursorSpecification`, you can specify setting the fetch sizes on its children and thereby determine the shape of the fetch block.

If a default fetch size is set on a `CursorSpecification`, you can use the `setFetchSize` method on the `Cursor` for that `CursorSpecification` to change the fetch size of the `Cursor`. By default, the root `CursorSpecification` of a `CursorManagerSpecification` has the fetch size set to 100.

Example: Specifying the fetch size and fetch block for a table view

This example creates a `Source` that represents the sales amount measure values as specified by selections of values from the dimensions of the measure. The product and customer selections each have ten values, the time selection has four values, and the promotion and channel selections each have one value. Assuming that a sales amount exists for each set of dimension values, the result set of the query has 300 elements (10*10*3*1*1).

To match a display of the elements that contains only twenty rows, the example sets a fetch size of twenty elements on the top-level `CompoundCursor`. Because the default fetch size is automatically set on the root `CursorSpecification`, which in this example is the `CompoundCursorSpecification` for the top-level `CompoundCursor`, the example just uses the `setFetchSize` method on the `CompoundCursor` to change the fetch size. The fetch block is the set of output and base values specified by twenty positions of the top-level `CompoundCursor`. The `TransactionProvider` is `tp` and the `DataProvider` is `dp`.

```
Source salesAmountsForSelections = salesAmount.join(customerSel)
                                              .join(productSel);
                                              .join(timeSel);
                                              .join(channelSel);
                                              .join(promotionSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a Cursor for salesAmountsForSelections
CursorManagerSpecification cursorMngrSpec =
     dp.createCursorManagerSpecification(salesAmountsForSelections);
SpecifiedCursorManager cursorMngr = dp.createCursorManager(cursorMngrSpec);
Cursor cursor = cursorMngr.createCursor();

// Set the fetch size of the top-level CompoundCursor to 20
cursor.setFetchSize(20);
```

Example: Using extents to specify the fetch sizes for a crosstab view

This example modifies the example in “Example: Navigating for a crosstab view without pages” on page 9-11. In this example, the number of times that the `for` loops are repeated depends upon the extent of the Cursor. As the conditional statement of the for loops, instead of specifying the number of positions that the Cursor has, this example gets the extent of the Cursor and uses the extent as the condition. The optimal fetch block for the crosstab display is a fetch block that contains, for each position of the CompoundCursor, the extent of the child Cursor elements at that position.

This example creates a CursorManagerSpecification and gets the root CursorSpecification. It casts the root CursorSpecification as a CompoundCursorSpecification. The example specifies setting the default fetch sizes on the children of the root CompoundCursorSpecification and it specifies the calculation of its extent.

The example sets the fetch size on each output ValueCursor equal to the extent of the ValueCursor. It then gets the displayable portion of the crosstab by looping through the positions of the child ValueCursor objects.

```
Source unitPriceByDay = unitPrice.join(productSel)
                                 .join(timeSel);
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create a CursorManagerSpecification for unitPriceByDay
CursorManagerSpecification cursorMngrSpec =
            dp.createCursorManagerSpecification(unitPriceByDay);
/*
 * Get the root CursorSpecification and cast it to a
 * CompoundCursorSpecification
 */
CompoundCursorSpecification rootSpec =
(CompoundCursorSpecification) cursorMngrSpec.getRootCursorSpecification();
/*
 * Specify setting the fetch size on the child Cursor objects
 * and calculating the extent of the positions in the Cursor
 */
rootSpec.specifyDefaultFetchSizeOnChildren();
rootSpec.setExtentCalculationSpecified(true);

// Create the CursorManager and the Cursor
SpecifiedCursorManager cursorMngr =
                             dp.createCursorManager(cursorMngrSpec);
Cursor unitPriceByDayCursor = cursorMngr.createCursor();

// Cast the Cursor to a CompoundCursor
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
/*
 * Determine a starting position and the number of rows to display.
 * The position in columnCursor at which the current display starts
 * is colStart and rowStart is the position in rowCursor at which
 * the current display starts.
 */
int colStart = 1;
int rowStart = 1;
String productValue;
String timeValue;
double price;
/*
 * The number of values from the ValueCursor objects for products and
 * days are now initialized as 1 because the ValueCursor objects have
 * at least one element.
 */
int numProducts = 1;
int numDays = 1;

// Get the ValueCursor and the outputs
CompoundCursor rootCursor = (CompoundCursor) unitPriceByDayCursor;
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);
ValueCursor unitPriceValues = rootCursor.getValueCursor();// Prices

// Loop through the positions of the faster varying output Cursor
for(int pPos = colStart; pPos < colStart + numProducts; pPos++) {
  columnCursor.setPosition(pPos);
  // Get the extents of the output ValueCursor objects
  numProducts = columnCursor.getExtent();
  numDays = rowCursor.getExtent();
  // Set the fetch sizes
  columnCursor.setFetchSize(numProducts);
  rowCursor.setFetchSize(numMonths);
  // Loop through the positions of the slower varying output Cursor
  for(int tPos = rowStart; tPos < rowStart + numDays; tPos++) {
    rowCursor.setPosition(tPos);
    /*
     * Get the values. Sending the values to the appropriate
     * display mechanism is not shown.
     */
    productValue = columnCursor.getCurrentString();
    timeValue = rowCursor.getCurrentString();
    price = unitPriceValues.getCurrentDouble();
  }
}
```

10 Understanding Cursor Classes and Concepts

Chapter summary

This chapter 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. For examples of creating and using a `Cursor` and its related objects, see Chapter 9.

List of topics

This chapter includes the following topics:

Overview of the OLAP API Cursor Objects

About the Cursor class and its related classes

A `Cursor` retrieves the result set defined by a `Source`. Creating a `Cursor` for a `Source` requires at least two intermediate steps. 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 on the `DataProvider` that you are using. The `CursorManagerSpecification` has `CursorSpecification` objects in a structure that mirrors the values and outputs of the `Source`.
2. Creating a `CursorManager` by calling the `createCursorManager` method on 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.
3. Creating a `Cursor` by calling the `createCursor` method on 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.

For an example of creating a `Cursor`, see Chapter 9.

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 of `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.

Sources for which you cannot create a Cursor

Some `Source` objects are not queries, which means the `Source` is either an incomplete specification or it does 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 has an unspecified input and is therefore an incomplete specification of the data to retrieve.
- 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 on `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 on the TransactionProvider your application is using. For information on Transaction and TransactionProvider objects, see Chapter 8.

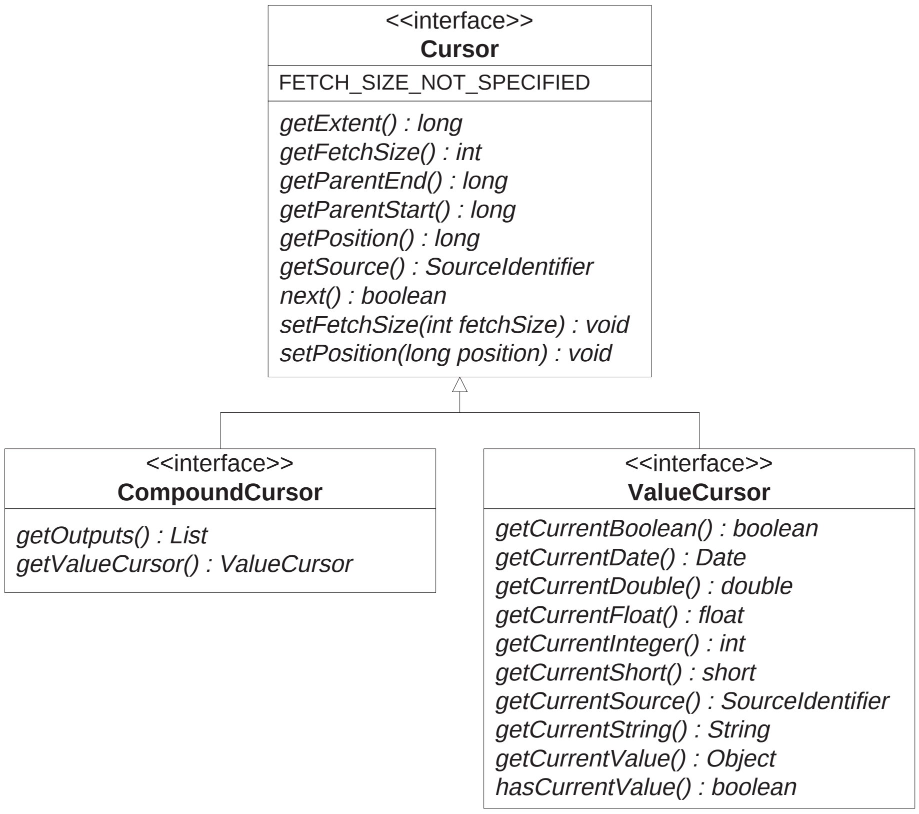
Cursor Class

Cursor class hierarchy

The Oracle OLAP API defines the following three interfaces in the oracle.olapi.data.cursor package:

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.

The following figure shows the class hierarchy of the Cursor classes. The CompoundCursor and ValueCursor interfaces extend the Cursor interface.


<<interface>>
Cursor
FETCH_SIZE_NOT_SPECIFIED
getExtent() : long
getFetchSize() : int
getParentEnd() : long
getParentStart() : long
getPosition() : long
getSource() : SourceIdentifier
next() : boolean
setFetchSize(int fetchSize) : void
setPosition(long position) : void
<<interface>>
CompoundCursor
getOutputs() : List
getValueCursor() : ValueCursor
<<interface>>
ValueCursor
getCurrentBoolean() : boolean
getCurrentDate() : Date
getCurrentDouble() : double
getCurrentFloat() : float
getCurrentInteger() : int
getCurrentShort() : short
getCurrentSource() : SourceIdentifier
getCurrentString() : String
getCurrentValue() : Object
hasCurrentValue() : boolean

Structure of a Cursor

The structure of a Cursor mirrors the structure of its Source. If the Source does not have any outputs, the Cursor for that Source is a ValueCursor. If the Source has one or more outputs, 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.

For example, suppose you have created a derived Source called productSel that represents a selection of product identification values from a primary Source that represents values from a dimension of products. You have selected 815, 1050, and 2055 as the values for productSel. If you create a Cursor for productSel, then that Cursor is a ValueCursor because productSel has no outputs.

You have also created a derived Source called timeSel that represents a selection of day values from a primary Source that represents a dimension of time values. The values of timeSel are 1-JAN-00, 1-APR-00, 1-JUL-00, and 1-OCT-00.

You have an MdmMeasure that represents values for the price of product units. The MdmMeasure has as inputs the MdmDimension objects representing products and times. You get a Source called unitPrice from the measure. The Source has products and times as inputs.

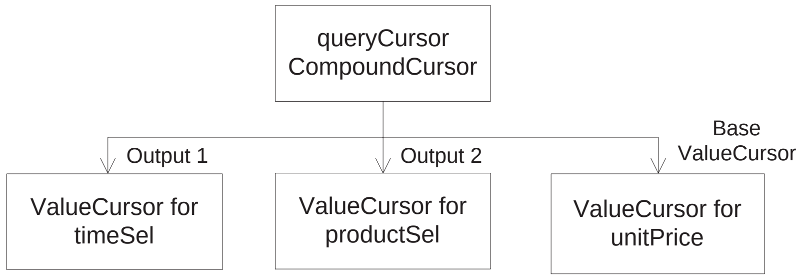
You join productSel and timeSel to unitPrice to create a Source, unitPriceByDay, which has productSel and timeSel as outputs, as in the following:

```
unitPriceByDay = unitPrice.join(productSel).join(timeSel);
```

The result set defined by unitPriceByDay is unit price values organized by the outputs. Since timeSel is joined to the result of unitPrice.join(productSel), 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 unitPriceByDay are the fastest varying of all, because there is one price value for each product for each day.

You then create a Cursor, queryCursor, for unitPriceByDay. Since unitPriceByDay 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 unitPriceByDay. The outputs for queryCursor are a ValueCursor that has values from productSel and a ValueCursor that has values from timeSel.

The following figure illustrates the structure of queryCursor. The top box represents queryCursor, which is the parent CompoundCursor. The bottom row of three boxes represents the three children of `queryCursor`, which are the base `ValueCursor` and the two output `ValueCursor` objects.



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 specified product on the specified day.

Day	Product	Price of Unit
01-JAN-00	815	58
01-JAN-00	1050	24
01-JAN-00	2055	24
01-APR-00	815	59
01-APR-00	1050	24
01-APR-00	2055	25
01-JUL-00	815	59
01-JUL-00	1050	25
01-JUL-00	2055	25
01-OCT-00	815	61
01-OCT-00	1050	25
01-OCT-00	2055	26

For examples of getting the values from a `ValueCursor`, see Chapter 9.

Specifying the behavior of a Cursor

The `CursorSpecification` objects of a `CursorManagerSpecification` specify some aspects of the behavior of their corresponding `Cursor` objects. You must specify the behavior on a `CursorSpecification` before creating the corresponding `Cursor`. If you have specified the behavior, you can successfully use the following `Cursor` methods:

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

Before you can use the `Cursor` methods listed above, you must specify the behavior by calling the following `CursorSpecification` methods:

- `setDefaultFetchSize`
- `setExtentCalculationSpecified`
- `setParentEndCalculationSpecified`
- `setParentStartCalculationSpecified`
- `specifyDefaultFetchSizeOnChildren`
  (for a `CompoundCursorSpecification` only)

A `CursorSpecification` also has methods you can use to discover if the behavior is specified. Those methods are the following:

- `isExtentCalculationSpecified`
- `isParentEndCalculationSpecified`
- `isParentStartCalculationSpecified`

For examples of specifying `Cursor` behavior, see Chapter 9. For information on fetch sizes, see “What is the fetch size of a Cursor?” on page 10-32. For information on the extent of a `Cursor`, see “What is the extent of a Cursor?” on page 10-30. 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 10-28.

Cursor methods

All `Cursor` objects have the following methods:

Method	Return Value
`getExtent`	The number of elements of the `Cursor`.
`getFetchSize`	The fetch size for the `Cursor`.
`getParentEnd`	The position of the parent `Cursor` at which the current value of the child `Cursor` ends.
`getParentStart`	The position of the parent `Cursor` at which the current value of the child `Cursor` starts.
`getPosition`	The position of the current element of the `Cursor`.
`getSource`	The `SourceIdentifier` for the `Cursor`.
`next`	A `boolean` that indicates whether an additional element exists in the `Cursor` and that the current position of the `Cursor` has advanced to that element.
`setFetchSize`	Void. This method specifies the fetch size for the `Cursor`.
`setPosition`	Void. This method sets the current position of the `Cursor`.

CompoundCursor methods

In addition to the methods inherited from `Cursor`, a `CompoundCursor` has the following methods:

Method	Return Value
`getOutputs`	A `List` that contains the outputs of the `CompoundCursor`.
`getValueCursor`	The base `ValueCursor` for the `CompoundCursor`.

ValueCursor methods

In addition to the methods inherited from `Cursor`, a `ValueCursor` has the following methods:

Method	Return Value
`getCurrentBoolean`	The `boolean` value at the current position.
`getCurrentDate`	The `Date` value at the current position.

Method	Return Value
getCurrentDouble	The double value at the current position.
getCurrentFloat	The float value at the current position.
getCurrentInteger	The int value at the current position.
getCurrentShort	The short value at the current position.
getCurrentSource	The Source at the current position.
getCurrentString	The String value at the current position.
getCurrentValue	The value at the current position.
hasCurrentValue	A boolean that indicates whether a value exists at the current position.

CursorManagerSpecification Class

About the 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.

The structure of a Cursor reflects the structure of its CursorManagerSpecification. A Cursor can be a single Cursor, for a Source with no outputs, or a Cursor 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, 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 if a CursorManagerSpecification has expired by calling the isExpired method on the CursorManagerSpecification.

CursorManagerSpecification Methods

A CursorManagerSpecification has the following methods:

Method	Return Value
getRootCursorSpecification	The top-level CursorSpecification.
getTransaction	The Transaction in which the CursorManagerSpecification was created.
isExpired	A boolean that indicates whether the CursorManagerSpecification has expired.

CursorSpecification Class

About the CursorSpecification class

A CursorSpecification specifies certain aspects of the behavior of the Cursor that corresponds to it. You do not create a CursorSpecification directly. 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, 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 the OLAP service should calculate the extent of a Cursor.
- Find out if calculating the extent is specified.

- Specify that the OLAP service 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.
- Find out if 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 10-22 and “About Fetch Sizes and Fetch Blocks” on page 10-32.

The Oracle OLAP API defines the following three classes in the `oracle.olapi.data.source` package:

`CursorSpecification`	An abstract superclass that implements methods inherited by its subclasses.
`ValueCursorSpecification`	A `CursorSpecification` for a `Source` that has values and no outputs.
`CompoundCursorSpecification`	A `CursorSpecification` for a `Source` that has one or more outputs. A `CompoundCursorSpecification` has component child `CursorSpecification` 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 on the corresponding `CursorSpecification` before it creates the `Cursor`.

CursorSpecification methods

All `CursorSpecification` objects have the following methods. The *set* methods specify the behavior of the `Cursor` that corresponds to the `CursorSpecification`.

Method	Return Value
`acceptVisitor`	An `Object`. This method accepts a `CursorSpecificationVisitor`. For more information, see the *Oracle9i OLAP Services OLAP API Reference*
`getDefaultFetchSize`	The fetch size set as the default on the `CursorSpecification`.
`getSource`	The `Source` for the `CursorSpecification`.
`isExtentCalculationSpecified`	A `boolean` that indicates whether the `CursorSpecification` specifies the calculation of the extent of the `Cursor`.
`isParentEndCalculationSpecified`	A `boolean` that indicates whether the `CursorSpecification` specifies the calculation of the ending position in the parent `Cursor` for the current value of the child `Cursor`.
`isParentStartCalculationSpecified`	A `boolean` that indicates whether the `CursorSpecification` specifies the calculation of the starting position in the parent `Cursor` for the current value of the child `Cursor`.
`setDefaultFetchSize`	Void. This method specifies a default fetch size for the `Cursor`. If a default fetch size is specified on a `CursorSpecification`, then you can change the fetch size on the `Cursor`.
`setExtentCalculationSpecified`	Void. This method specifies whether to calculate the extent of the `Cursor`.

Method	Return Value
`setParentEndCalculationSpecified`	Void. This method specifies whether to calculate the ending position in the parent `Cursor` for the current value of the `Cursor`.
`setParentStartCalculationSpecified`	Void. This method specifies whether to calculate the starting position in the parent `Cursor` for the current value of the `Cursor`.

CompoundCursorSpecification methods

In addition to the methods it inherits from `CursorSpecification`, a `CompoundCursorSpecification` has the following methods:

Method	Return Value
`acceptVisitor`	An `Object`. This method accepts a `CursorSpecificationVisitor`.
`getOutputs`	A `List` of the `CursorSpecification` objects that are the outputs of this `CompoundCursorSpecification`.
`getValueCursorSpecification`	The `ValueCursorSpecification` for this `CompoundCursorSpecification`.
`specifyDefaultFetchSizeOnChildren`	Void. This method specifies that the default fetch size is set on the child `CursorSpecification` objects of this `CompoundCursorSpecification`. This method has two versions, one that supplies a fetch size and one that does not.

ValueCursorSpecification methods

In addition to the methods it inherits from `CursorSpecification`, a `ValueCursorSpecification` has an override of the following method:

Method	Return Value
`acceptVisitor`	An `Object`. This method accepts a `CursorSpecificationVisitor`.

CursorManager Class

About the CursorManager class

A CursorManager manages the buffering of data for the Cursor objects it creates. To create a CursorManager, call the createCursorManager method on a DataProvider and pass it a CursorManagerSpecification.

You can create more than one Cursor from the same CursorManager, which is useful for displaying data from a result set in different formats such as a table or a graph. All of the Cursor objects created by a CursorManager have the same specifications, such as the default fetch sizes and the levels at which fetch sizes are set. Because the Cursor objects have the same specifications, they can share the data managed by the CursorManager.

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 if the SpecifiedCursorManager is open, and for closing it. The createCursorManager method on DataProvider returns 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 the OLAP service. 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 on the existing CursorManager to replace the previous CursorManagerSpecification with the new CursorManagerSpecification. You can then create a new Cursor from the CursorManager.

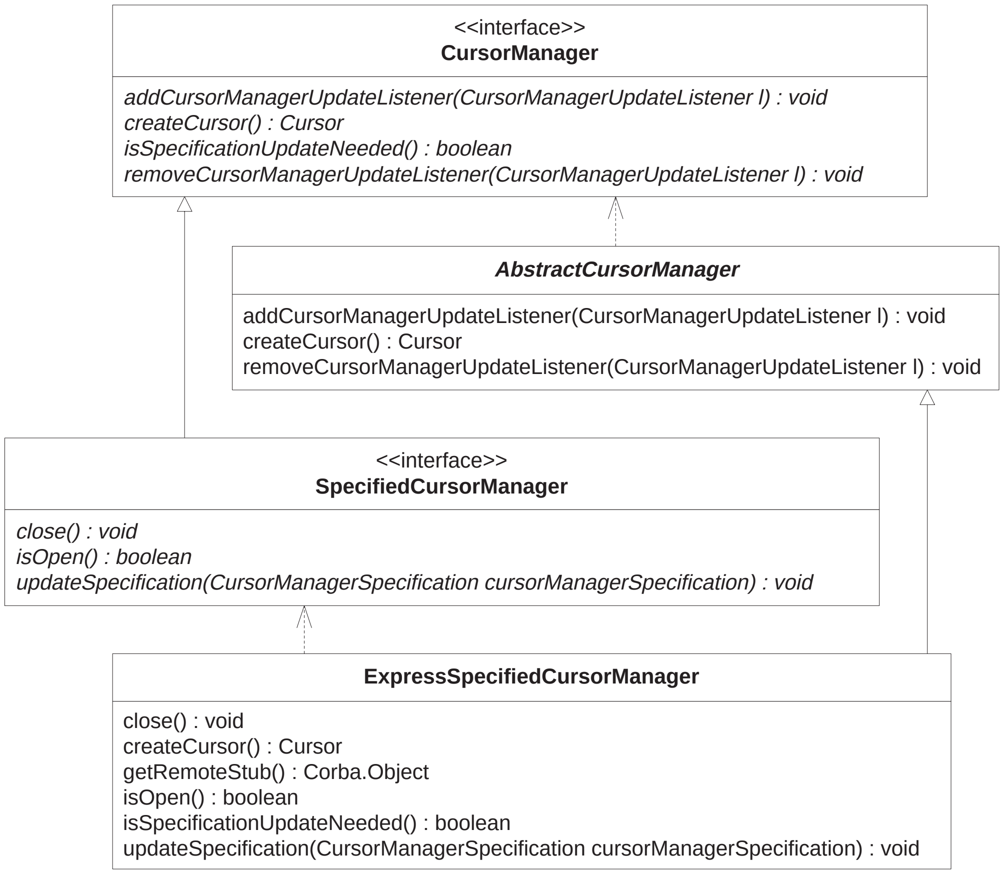
To find out if the `CursorManagerSpecification` for a `CursorManager` needs updating, call the `isSpecificationUpdateNeeded` method on the `CursorManager`. You can also use a `CursorManagerUpdateListener` to listen for events generated by changes in a `Source`. For more information, see "CursorManagerUpdateListener Class" on page 10-20.

CursorManager class hierarchy

The following table lists most of the `CursorManager` interfaces and classes:

`CursorManager`	An interface that has defines methods for all `CursorManager` objects.
`AbstractCursorManager`	A `CursorManager` that implements methods for adding and removing `CursorManagerUpdateListener` objects. For more information, see "CursorManagerUpdateListener Class" on page 10-20.
`SpecifiedCursorManager`	An interface that defines additional methods for a `CursorManager`.
`ExpressSpecifiedCursorManager`	A class that implements the `SpecifiedCursorManager` interface and extends `AbstractCursorManager`. In the Oracle OLAP API, the `createCursorManager` method on `DataProvider` returns an instance of this class.

The following figure shows the relationships of the `CursorManager` classes described in the preceding table. A solid line and a closed arrowhead indicate that a class extends the class to which the arrow points. A dotted line and an open arrowhead indicate that the class implements the interface to which the arrow points.


<<interface>>
CursorManager
addCursorManagerUpdateListener(CursorManagerUpdateListener l) : void
createCursor() : Cursor
isSpecificationUpdateNeeded() : boolean
removeCursorManagerUpdateListener(CursorManagerUpdateListener l) : void
AbstractCursorManager
addCursorManagerUpdateListener(CursorManagerUpdateListener l) : void
createCursor() : Cursor
removeCursorManagerUpdateListener(CursorManagerUpdateListener l) : void
<<interface>>
SpecifiedCursorManager
close() : void
isOpen() : boolean
updateSpecification(CursorManagerSpecification cursorManagerSpecification) : void
ExpressSpecifiedCursorManager
close() : void
createCursor() : Cursor
getRemoteStub() : Corba.Object
isOpen() : boolean
isSpecificationUpdateNeeded() : boolean
updateSpecification(CursorManagerSpecification cursorManagerSpecification) : void

CursorManager methods

All CursorManager objects have the following methods:

Method	Return Value
addCursorManagerUpdateListener	Void. This method adds a CursorManagerUpdateListener to the CursorManager.
createCursor	A Cursor whose structure is specified by the CursorManagerSpecification of the CursorManager.
isSpecificationUpdateNeeded	A boolean that indicates whether the CursorManager needs to update its CursorManagerSpecification.
removeCursorManagerUpdateListener	Void. This method removes a CursorManagerUpdateListener from the CursorManager.

AbstractCursorManager methods

An AbstractCursorManager implements the following methods of the CursorManager interface:

Method	Return Value
addCursorManagerUpdateListener	Void. This method adds a CursorManagerUpdateListener to the CursorManager.
removeCursorManagerUpdateListener	Void. This method removes a CursorManagerUpdateListener from the CursorManager.

SpecifiedCursorManager methods

A `SpecifiedCursorManager` has the following methods:

Method	Return Value
`close`	Void. This method closes the `SpecifiedCursorManager` and releases the resources associated with it.
`isOpen`	A `boolean` that indicates whether the `SpecifiedCursorManager` is open.
`updateSpecification`	Void. This method replaces the `CursorManagerSpecification` of the `SpecifiedCursorManager`.

ExpressSpecifiedCursorManager methods

In addition to the methods it inherits from `AbstractCursorManager`, an `ExpressSpecifiedCursorManager` implements the following methods:

Method	Return Value
`close`	Void. This method closes the `ExpressSpecifiedCursorManager` and releases the resources associated with it.
`createCursor`	A `Cursor`.
`getRemoteStub`	A stub for a CORBA object that an application can use in constructing a new `ReceiveOnlyCursorManager`.
`isOpen`	A `boolean` that indicates whether the `ExpressSpecifiedCursorManager` is open.
`isSpecificationUpdateNeeded`	A `boolean` that indicates whether the `ExpressSpecifiedCursorManager` needs to update its `CursorManagerSpecification`.
`updateSpecification`	Void. This method replaces the `CursorManagerSpecification` for the `ExpressSpecifiedCursorManager`.

CursorManagerUpdateListener Class

About the CursorManagerUpdateListener class

`CursorManagerUpdateListener` is an interface that has methods that receive `CursorManagerUpdateEvent` objects. Oracle OLAP Services 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 on 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`.

CursorManagerUpdateListener methods

A `CursorManagerUpdateListener` has the following methods:

Method	Return Value
`cursorManagerDataUpdated`	Void. Called by a `CursorManager` when it becomes aware that its `Source` has changed so that the data specified by `Source` is different but the structure of the `Source` has not changed.
`cursorManagerSpecificationUpdated`	Void. Called by a `CursorManager` when its `CursorManagerSpecification` has been updated.
`cursorManagerStructureUpdated`	Void. Called by a `CursorManager` when it becomes aware that the structure of its `Source` has changed.

About the CursorManagerUpdateEvent class

Oracle OLAP Services 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 Services 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.

CursorManagerUpdateEvent fields

A `CursorManagerUpdateEvent` has the following fields:

Field	Meaning
`CURSOR_MANAGER_DATA_UPDATED`	Indicates that the `Source` for a `CursorManager` has changed so that the data specified by `Source` is different but the structure of the `Source` has not changed
`CURSOR_MANAGER_SPECIFICATION_UPDATED`	Indicates that the `CursorManager` has updated its `CursorManagerSpecification`.
`CURSOR_MANAGER_STRUCTURE_UPDATED`	Indicates that the `Source` for a `CursorManager` has changed and the structure of the `Source` is different.

CursorManagerUpdateEvent methods

A `CursorManagerUpdateEvent` has the following methods:

Method	Return Value
`getCursorManager`	The `CursorManager` that originated the event.
`getID`	A constant the identifies the type of event that occurred, such as `CURSOR_MANAGER_STRUCTURE_UPDATED`.

About Cursor Positions and Extent

About positions of a Cursor

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 on the `Cursor`.

Oracle OLAP Services 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, the `Cursor` throws a `PositionOutOfBoundsException`.

Positions of a ValueCursor

The current position of a `ValueCursor` specifies a value, which you can retrieve. For example, `productSel`, a derived `Source` described in "Structure of a Cursor" on page 10-5, is a selection of three products from a primary `Source` that specifies a dimension of products and their hierarchical groupings. The `ValueCursor` for `productSel` has three elements. The following example gets the position of each element of the `ValueCursor`, and displays the value at that position. The `output` object is a `PrintWriter`.

```
// productSelValCursor is the ValueCursor for productSel
do {
  output.print(productSelValCursor.getPosition + " : ");
  output.println(productSelValCursor.getCurrentValue);
}
while(productSelValCursor.next();
```

The above example displays the following:

```
1 : 815
2 : 1050
3 : 2055
```

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

```
productSelValCursor.setPosition(2);
output.println(productSelValCursor.getCurrentValue);
```

The above example displays the following:

```
1050
```

For more examples of getting the current value of a `ValueCursor`, see Chapter 9.

Positions of a CompoundCursor

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

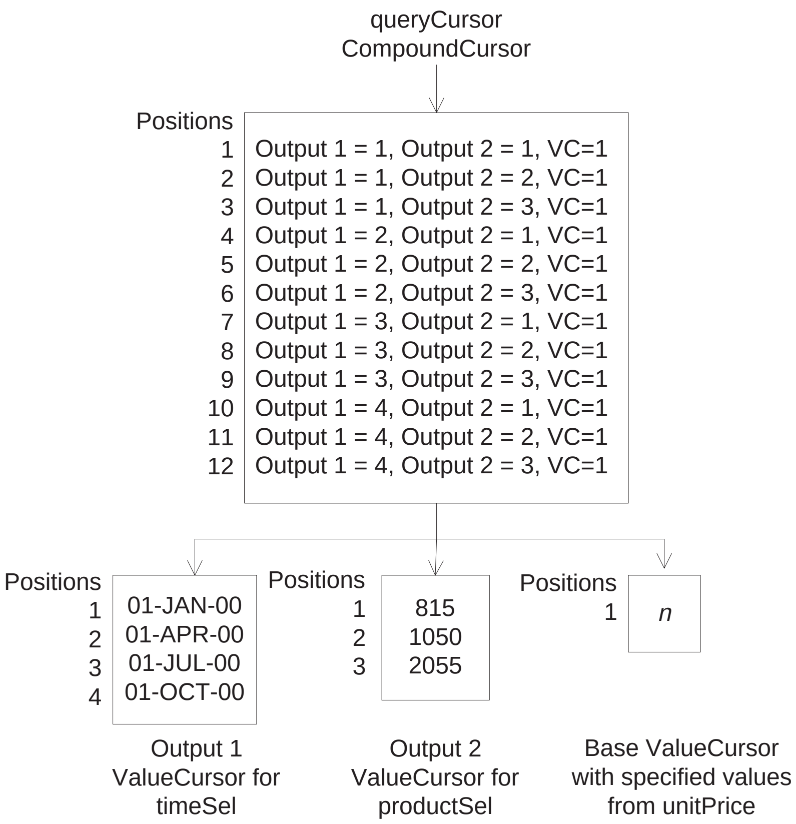
For example, `unitPriceByDay`, the `Source` described in “Structure of a Cursor” on page 10-5, has values from a measure, `unitPrice`. The values are the prices of product units at different times. The outputs of `unitPriceByDay` are `Source` objects that represent selections of four day values from a time dimension and three product values from a product dimension.

The result set for `unitPriceByDay` 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 days). Therefore, the `queryCursor` `CompoundCursor` created for `unitPriceByDay` 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 `productSel`)
- 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.)

The following figure illustrates the `queryCursor` `CompoundCursor` with its base `ValueCursor` and its outputs. Each `Cursor` is represented by a box. The positions of the elements of the `Cursor` appear outside the box. For the child `Cursor` objects, the values at those positions appear inside the box. For the `CompoundCursor`, the set of positions of its `ValueCursor` objects that are specified at each position appear inside the box.


queryCursor
CompoundCursor
Positions
1 Output 1 = 1, Output 2 = 1, VC=1
2 Output 1 = 1, Output 2 = 2, VC=1
3 Output 1 = 1, Output 2 = 3, VC=1
4 Output 1 = 2, Output 2 = 1, VC=1
5 Output 1 = 2, Output 2 = 2, VC=1
6 Output 1 = 2, Output 2 = 3, VC=1
7 Output 1 = 3, Output 2 = 1, VC=1
8 Output 1 = 3, Output 2 = 2, VC=1
9 Output 1 = 3, Output 2 = 3, VC=1
10 Output 1 = 4, Output 2 = 1, VC=1
11 Output 1 = 4, Output 2 = 2, VC=1
12 Output 1 = 4, Output 2 = 3, VC=1
Positions
1 01-JAN-00
2 01-APR-00
3 01-JUL-00
4 01-OCT-00
Positions
1 815
2 1050
3 2055
Positions
1 n
Output 1 ValueCursor for timeSel
Output 2 ValueCursor for productSel
Base ValueCursor with specified values from unitPrice

Note: 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 `unitPriceByDay`, 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`.

The following figure 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 day values. In the top row are the product values. In each of the intersecting cells of the crosstab is the price of the product on the day.

	815	1050	2055
01-JAN-00	58	24	24
01-APR-00	59	24	25
01-JUL-00	59	25	25
01-OCT-00	61	25	26

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. For example, the following operations set the position of `queryCursor` and then get the current values and the positions of the child `Cursor` objects.

```
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;
root.setPosition(pos);
System.out.println("CompoundCursor position set to " + pos + ".");
System.out.println("CC position = " + 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());
```

The above example displays the following:

```
CompoundCursor position set to 5.
CC position = 5.
Output 1 position = 2, value = 01-APR-00
Output 2 position = 2, value = 1050
VC position = 1, value = 24
```

The positions of `queryCursor` are symmetric in that the result set for `unitPriceByDay` always has three product values for each time value. The `ValueCursor` for `productSel`, therefore, always has three positions for each value of the timeSel ValueCursor. The timeSel output ValueCursor is slower varying than the productSel 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 2055 on October 1, 2000 were null because that product was no longer being purchased by that date, and if null values were suppressed in the query, then queryCursor would only have eleven positions. The ValueCursor for productSel would only have two positions when the position of the ValueCursor for timeSel was 4.

Example: Positions in an asymmetric query

This example of an asymmetric result set is produced by revising the query from "Structure of a Cursor" on page 10-5 as follows.

```
productByPriceOnDay = productSel.join(unitPrice).join(timeSel);
```

Now the result of the query is the products by price on a day. The base values of productByPriceOnDay are the values from productSel as specified by the values of unitPrice and timeSel.

Because productByPriceOnDay 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 productByPriceOnDay, loops through the positions of the CompoundCursor, gets the position and current value of each child ValueCursor object, and displays the positions and values.

```
 //Prepare and commit the current Transaction.
try{
  tp.prepareCurrentTransaction();
}
catch(NotCommittableException e){
  output.println("Caught exception " + e + ".");
}
tp.commitCurrentTransaction();

// Create the Cursor. The DataProvider is dp.
CursorManagerSpecification cursorMngrSpec =
                  dp.createCursorManagerSpecification(productByPriceOnDay);
CursorManager cursorManager = dp.createCursorManager(cursorMngrSpec);
Cursor queryCursor2 = cursorManager.createCursor();

// Get the ValueCursor and the outputs
```

```
CompoundCursor rootCursor = (CompoundCursor) queryCursor2;
ValueCursor baseValueCursor = rootCursor.getValueCursor();
List outputs = rootCursor.getOutputs();
ValueCursor output1 = (ValueCursor) outputs.get(0);
ValueCursor output2 = (ValueCursor) outputs.get(1);

// Get the positions and values and display them.
System.out.println("   CC \t\tOutput 1 \tOutput 2 \tVC");
System.out.println("position \tposition:value " +
                   "\tposition:value \tposition:value");
do {
System.out.println("      " + root.getPosition() +
                   "\t\t   " + output1.getPosition() +
                   "  :  " + output1.getCurrentValue() +
                   "\t   " + output2.getPosition() +
                   "  :  "  + output2.getCurrentValue() +
                   "\t   " + baseValueCursor.getPosition() +
                   "  :  " + baseValueCursor.getCurrentValue());
}
while(queryCursor2.next();
```

This example displays the following:

```
   CC        Output 1           Output 2           VC
position     position:value     position:value     position:value
    1          1  :  01-JAN-00      1  :  58           1  :  815
    2          1  :  01-JAN-00      2  :  24           1  :  1050
    3          1  :  01-JAN-00      2  :  24           2  :  2055
    4          2  :  01-APR-00      1  :  59           1  :  815
    5          2  :  01-APR-00      2  :  24           1  :  1050
    6          2  :  01-APR-00      3  :  25           1  :  2055
    7          3  :  01-JUL-00      1  :  59           1  :  815
    8          3  :  01-JUL-00      2  :  25           1  :  1050
    9          3  :  01-JUL-00      2  :  25           2  :  2055
    10         4  :  01-OCT-00      1  :  61           1  :  815
    11         4  :  01-OCT-00      2  :  25           1  :  1050
    12         4  :  01-OCT-00      3  :  26           1  :  2055
```

The `ValueCursor` with `unitPrice` values (output 2) has only two positions for 01-JAN-00 and 01-JUL-00 because it has only two different values for those days. The prices of two of the products are the same on those two days: 24 for products 1050 and 2055 on January 1, 2000 and 25 for those same two products on July 1, 2000. The base `ValueCursor` for `queryCursor2` has two positions when the `timeSel` value is 01-JAN-00 or 01-JUL-00 because each of the `unitPrice` values for those days is not unique.

About the parent starting and ending positions in a Cursor

To effectively manage the display of the data 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 find out 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 “What is the fetch size of a Cursor?” on page 10-32.

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 sales amounts greater than $5,000 purchased by customers in certain cities during the first three months of the calendar year 2000. The products were sold through the direct sales channel (S) during a television promotion (TV).

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 promotion values
- output 2, a `ValueCursor` for the channel values
- output 3, 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 sales amounts over $5,000.

The following figure illustrates the parent, `cubeCursor`, with the values of its child `Cursor` objects layered horizontally. The slowest varying output, with the promotion values, 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 7 and 9 of `cubeCursor`, which is shown within the bold border. The positions, 1 through 10, of `cubeCursor` appear above the top row.

1	2	3	4	5	6	7	8	9	10
TV									
S									
2000-01			2000-02				2000-03		
Bonn		London	Bonn		London	Paris	Bonn	London	
1050	2055	815	1050	1555	935	1050	935	1050	3690

The current value of the output `ValueCursor` for the time `Source` is 2000-02. You cannot determine from the data within the block that the starting and ending positions of the current value, 2000-02, in the parent, `cubeCursor`, are 4 and 7, respectively.

The `cubeCursor` from the previous figure is shown again in the following figure, 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 2000-02 is (7 - 4 + 1) = 4.

1	2	3	4	5	6	7	8	9	10
1-10									
1-10									
1 to 3			4 to 7				8 to 10		
1 to 2		3 to 3	4 to 5		6 to 6	7 to 7	8 to 8	9 to 10	
1 to 1	2 to 2	3 to 3	4 to 4	5 to 5	6 to 6	7 to 7	8 to 8	9 to 9	10 to 10

To specify that you want the OLAP service to calculate the starting and ending positions of a value of a child `Cursor` in its parent `Cursor`, call the `setParentStartCalculationSpecified` and `setParentEndCalculationSpecified` methods on 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 on the `CursorSpecification`. For an example of specifying these calculations, see Chapter 9.

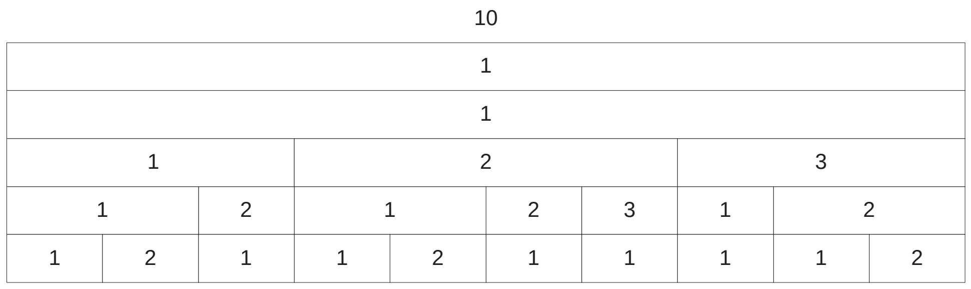
What is the extent of a Cursor?

The extent of a `Cursor` is the total number of elements it contains relative to any slower varying outputs. The following figure 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 10 so the extent of `cubeCursor` is therefore 10. That number is above the top row of the figure. The top row is the `ValueCursor` for the promotion value and the next row down is the `ValueCursor` for the channel value. The extent of each of those `ValueCursor` objects is 1 because they each have only one value.

The third row down represents the time values. Its extent is 3, since there are 3 months values. The next row down is the `ValueCursor` for the customers by city. 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 2, for the second month it is 3, and for the third month it is 2.

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 two products values are specified by the first set of month and city values (1050 and 2055 for Bonn in 2000-01), the extent of the products `ValueCursor` for that set is 2. For the second set of values for customers and times (2000-10, London), the extent of the products `ValueCursor` is 1, and so on.



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`, the OLAP service must traverse the entire result space before it creates the `CompoundCursor`. If you do not specify that you wants the extent calculated, then the OLAP service 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 the OLAP service to calculate the extent for a `Cursor`, call the `setExtentCalculationSpecified` method on the `CursorSpecification` corresponding to the `Cursor`. You can determine whether calculating the extent is specified by calling the `isExtentCalculationSpecified` method on the `CursorSpecification`. For an example of specifying the calculation of the extent of a `Cursor`, see Chapter 9.

About Fetch Sizes and Fetch Blocks

What is the fetch size of a Cursor?

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 the OLAP service. A `CursorManager` manages a virtual `Cursor` and retrieves results from the OLAP service as your application needs it. By managing the virtual `Cursor`, the `CursorManager` relieves your application of a substantial burden.

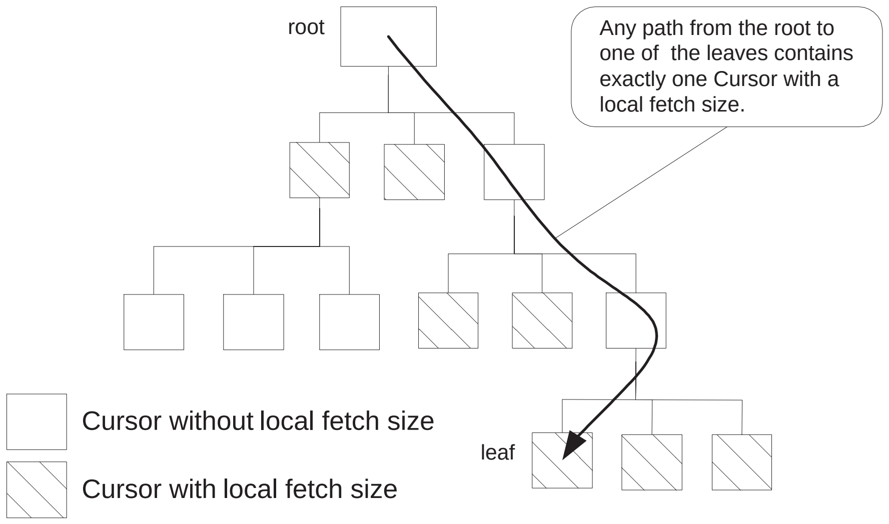
The amount of data that a `Cursor` retrieves in a single fetch operation is determined by the *fetch size* specified for the `Cursor`. For a `CompoundCursor`, the amount of data fetched in a single operation is the product of the fetch sizes of all of its descendent `ValueCursor` objects. The total set of values retrieved in a single fetch is the *fetch block* for the `Cursor`. You specify fetch sizes in order 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 of the data.

When you create a `CursorManagerSpecification` for a `Source`, as the first step in creating a `Cursor`, the OLAP service specifies a default fetch size on the root `CursorSpecification` of the `CursorManagerSpecification`. By calling methods on the `CursorSpecification` objects of the `CursorManagerSpecification`, you can specify a default fetch size or specify setting the fetch size at other levels of a `CompoundCursor`.

If the fetch size is specified on a `CursorSpecification`, then you can get or set the fetch size for the corresponding `Cursor` by calling the `getFetchSize` or `setFetchSize` method on that `Cursor`. For a `CompoundCursor`, you can set different fetch sizes for child `Cursor` objects at different levels in the outputs.

A `Cursor` has a *local fetch size* if the size of the fetch block is specified for that `Cursor`. Not all of the `Cursor` objects in a `CompoundCursor` can have local fetch sizes. The structure of a `CompoundCursor` is like a tree, with the hierarchy of `Cursor` objects starting at the topmost (root) `Cursor` and going down through all the child `Cursor` objects. Any path through the hierarchy, starting from the root and going down to a leaf `ValueCursor`, can contain one, and only one, `Cursor` with a local fetch size. Specifying the fetch size on a parent `Cursor` affects all of the child `Cursor` objects of that parent. This means that a fetch block can contain no more than the number of elements of each child `Cursor` specified by the fetch size.

The following figure shows an example of a path through the hierarchy of a `Cursor` tree in which the `Cursor` objects with local fetch sizes are shaded.


root
Any path from the root to one of the leaves contains exactly one Cursor with a local fetch size.
leaf
Cursor without local fetch size
Cursor with local fetch size

About determining the shape of a fetch block

In a `CompoundCursor`, the levels at which you set the fetch sizes determine the *shape* of the fetch block of the `CompoundCursor`. The optimal fetch block for a `CompoundCursor` depends on the way you intend to navigate the `Cursor` and display the data. After determining how to display the data, you should do the following:

- Specify a fetch block that is large enough to contain all the data required for the portion of the result set that you are displaying in the user interface. For example, if you display the data in a table and the size of the window means that 25 rows are visible at a time, then the fetch block should contain at least 25 rows. If it is any smaller than this, the `Cursor` needs to make multiple trips to the OLAP service to fill the display.
- Specify fetch sizes on the `Cursor` objects that you use to loop through the result set. For example, for a table view, set fetch sizes on the root `Cursor` and for a crosstab view, set fetch sizes on the child `Cursor` objects.

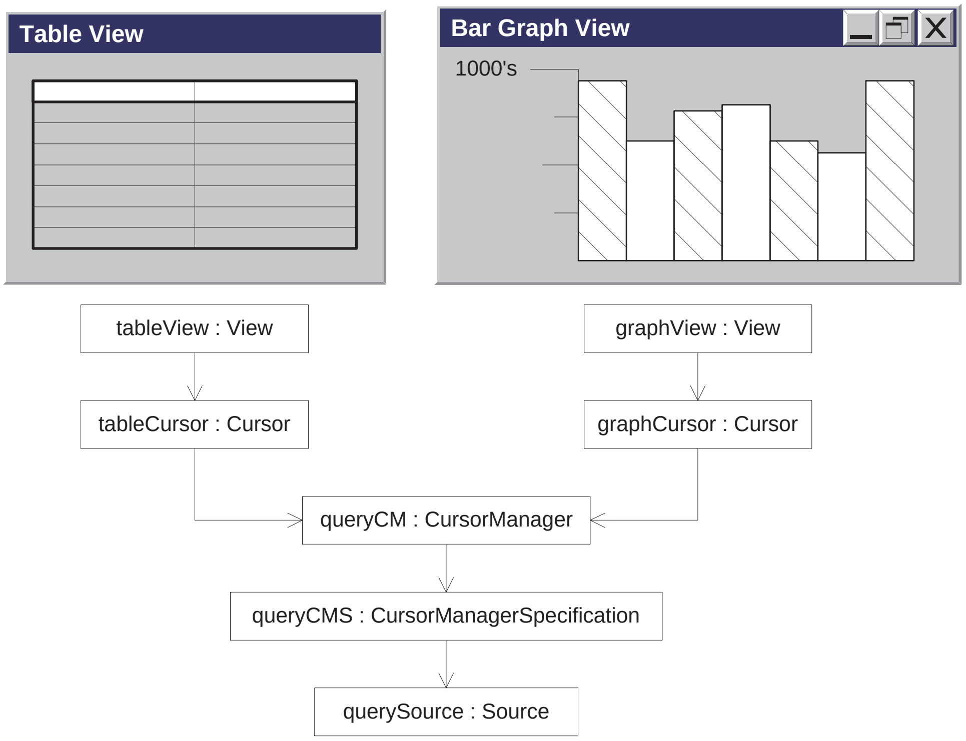
- Keep the product of all of the fetch sizes relatively small because the product determines the total number of cells in the fetch block. If the product of all the fetch sizes is too large, then you lose the advantages of the virtual `Cursor`.

For examples of specifying fetch sizes and fetch blocks for different displays, see Chapter 9.

About sharing fetch blocks

You can create two or more `Cursor` objects from the same `CursorManager` and use both `Cursor` objects simultaneously. The `Cursor` objects can share the data managed by the `CursorManager`, rather than having separate data caches, because the shape of the fetch blocks is the same for both `Cursor` objects. The shape of the fetch blocks is determined by the levels of the `CursorManagerSpecification` on which the fetch size is specified.

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. The following figure illustrates the relationship between the `Source`, the `Cursor` objects, and the views.



11

Creating Dynamic Queries

Chapter summary

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.

List of topics

This chapter includes the following topics:

- About Template Objects
- Overview of Template and Related Classes
- Example of a Template

About Template Objects

Features of 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 below. 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.

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` gets 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 on 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 the OLAP service. 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

What classes do I use to create a Template?

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

- DataProvider
- DynamicDefinition

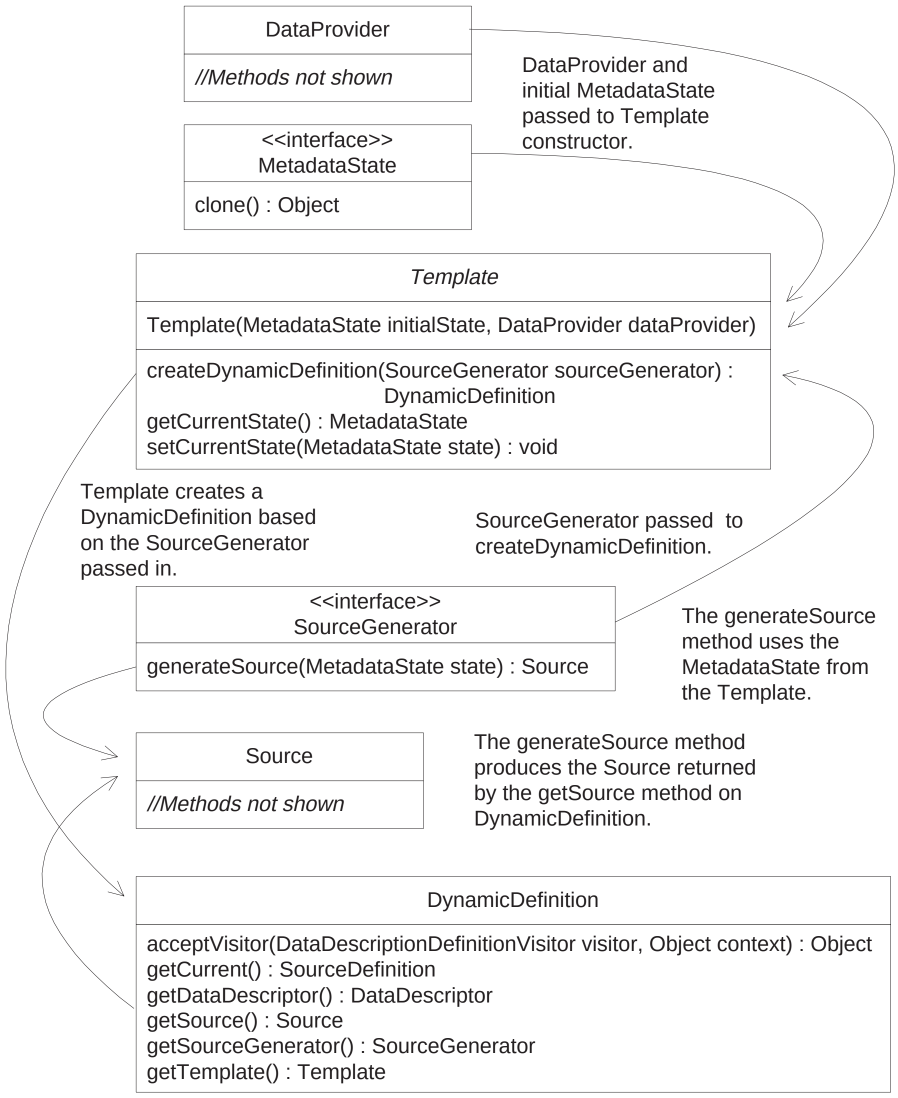
What is the relationship between the classes that produce a dynamic Source?

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 on the DynamicDefinition, the MetadataState is passed to the generateSource method on 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 on 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.

The following figure illustrates the relationship of the classes described in the preceding list. The arrows on the right indicate that the DataProvider and MetadataState objects are passed to the Template constructor and that the SourceGenerator is passed to the createDynamicDefinition method on the Template. The arrows on the left indicate that a DynamicDefinition is returned by the createDynamicDefinition method and that the same Source is returned by the generateSource method on the SourceGenerator and the getSource method on the DynamicDefinition.


DataProvider
//Methods not shown
<<interface>>
MetadataState
clone() : Object
DataProvider and initial MetadataState passed to Template constructor.
Template
Template(MetadataState initialState, DataProvider dataProvider)
createDynamicDefinition(SourceGenerator sourceGenerator) : DynamicDefinition
getCurrentState() : MetadataState
setCurrentState(MetadataState state) : void
Template creates a DynamicDefinition based on the SourceGenerator passed in.
SourceGenerator passed to createDynamicDefinition.
<<interface>>
SourceGenerator
generateSource(MetadataState state) : Source
The generateSource method uses the MetadataState from the Template.
Source
//Methods not shown
The generateSource method produces the Source returned by the getSource method on DynamicDefinition.
DynamicDefinition
acceptVisitor(DataDescriptionDefinitionVisitor visitor, Object context) : Object
getCurrent() : SourceDefinition
getDataDescriptor() : DataDescriptor
getSource() : Source
getSourceGenerator() : SourceGenerator
getTemplate() : Template

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`.

The `Template` abstract class implements the following methods:

Method	Return Value
`createDynamicDefinition`	A `DynamicDefinition` that is a proxy for the `SourceDefinition` that is immutably paired to the `Source` generated by the `SourceGenerator` passed to this method.
`getCurrentState`	The current state of the `MetadataState` for the `Template`.
`setCurrentState`	Void. This method specifies the `MetadataState` passed in as the current state for the `Template`.

For an example of a `Template` implementation, see “Example: Implementing a Template” on page 11-9.

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 on 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: Implementing a MetadataState” on page 11-12.

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 on the Template. You then get the Source by calling the getSource method on the DynamicDefinition.

A Template can create more than one DynamicDefinition, each with a differently implemented SourceGenerator. The generateSource methods on 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: Implementing a SourceGenerator" on page 11-13.

DynamicDefinition class

DynamicDefinition is a subclass of SourceDefinition. You create a DynamicDefinition by calling the createDynamicDefinition method on a Template and passing it a SourceGenerator. You get the Source produced by the SourceGenerator by calling the getSource method on 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: Getting the Source produced by the Template" on page 11-14.

Example of a Template

Designing the 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 sales amount as the measure and products as the base dimension in the first dialog box. From the Single Selections dialog box, the end user selects customers from San Francisco, the first quarter of 2000, the direct channel, and billboard promotions 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 sales amount values of those sold through the direct sales channel to customers from San Francisco during the first calendar quarter of the year 2000 while a billboard promotion was occurring.

For examples of implementations of the `TopBottomTemplate`, `TopBottomTemplateState`, and `TopBottomTemplateGenerator` objects, and an example of an application that uses them, see “Example: Implementing a Template” on page 11-9, “Example: Implementing a MetadataState” on page 11-12, “Example: Implementing a SourceGenerator” on page 11-13, and “Example: Getting the Source produced by the Template” on page 11-14.

Example: Implementing a Template

The following is an implementation of the `TopBottomTemplate` class described in “Designing the Template” on page 11-8.

```
package myTestPackage;

import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.DynamicDefinition;
import oracle.olapi.data.source.Source;
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 {
  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 default type and number values
   * and a 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 values 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: Implementing a MetadataState

The following is an implementation of the `TopBottomTemplateState` class described in "Designing the Template" on page 11-8.

```
package myTestPackage;

import oracle.olapi.data.source.Source;
import oracle.olapi.transaction.metadataStateManager.MetadataState;

/**
 * Stores data that can be changed by its TopBottomTemplate.
 * The data is used by a TopBottomTemplateGenerator in producing
 * a Source for the TopBottomTemplate.
 */
public 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: Implementing a SourceGenerator

The following is an implementation of the `TopBottomTemplateGenerator` class described in “Designing the Template” on page 11-8.

```
package myTestPackage;

import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.SourceGenerator;
import java.lang.Math;

/**
 * Produces a Source for a TopBottomTemplate based on the data
 * values of a TopBottomTemplateState.
 */
public 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));
  }
}
```

Example: Getting the Source produced by the Template

After you have stored the selections made by the end user in the `MetadataState` for the `Template`, use the `getSource` method on 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: Implementing a Template” on page 11-9. For brevity, the code does not contain much exception handling.

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

- Connects to an OLAP service.
- Opens a database.
- Gets metadata objects for the measure and the dimensions selected by the end user.
- Gets primary `Source` objects from the metadata objects.

The example does the following:

- Gets primary `Source` objects from the `Context`.

- 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.
- Creates a Cursor for that Source.
- Gets the values from the Cursor and displays them.

The following example does not include the code for interacting with the end user or for implementing the SingleSelectionTemplate or the MetadataState and SourceGenerator objects for the SingleSelectionTemplate. The example class has a method for creating a Cursor and a method for printing the values of the Cursor. All other operations occur in the main method. The Context object supplies the connection to the database, the DataProvider and the TransactionProvider, and primary Source objects.

```
package myTestPackage;

import oracle.olapi.data.source.Source;
import oracle.olapi.data.source.StringSource;
import oracle.olapi.data.source.DataProvider;
import oracle.olapi.data.source.CursorManagerSpecification;
import oracle.olapi.data.cursor.CursorManager;
import oracle.olapi.data.source.SpecifiedCursorManager;
import oracle.olapi.data.cursor.Cursor;
import oracle.olapi.data.cursor.ValueCursor;
import oracle.olapi.transaction.NotCommittableException;
import myTestPackage.Context;
import myTestPackage.TopBottomTemplate;
import myTestPackage.SingleSelectionTemplate;

/**
 * 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 {
  /**
   * Prints the values of the Cursor.
   */
  public static void printCursor(Cursor cursor) {
```

```
  /*
   * Because the result is a single set of values with no outputs,
   * cast the Cursor to a ValueCursor and print out the values.
   */
  ValueCursor valueCursor = (ValueCursor) cursor;
  int i = 1;
  do {
    System.out.println(i + ". " + valueCursor.getCurrentValue());
    i++;
  } while(valueCursor.next());
}

/**
 * Creates a Cursor.
 */
public static void createCursor(Source choice, DataProvider dp) {
  CursorManagerSpecification cursorMngrSpec =
                      dp.createCursorManagerSpecification(choice);
  SpecifiedCursorManager cursorManager =
                      dp.createCursorManager(cursorMngrSpec);
  Cursor cursor = cursorManager.createCursor();
  // Print the values of the Cursor.
  printCursor(cursor);
  // Close the CursorManager.
  cursorManager.close();
}

public static void main(String[] args) {
  /*
   * Create a Context object and from it get the DataProvider and
   * the primary Source objects for the measure and the dimensions.
   */
  Context context = new Context();
  DataProvider dp = context.getDataProvider();
  Source[] sources = context.getPrimarySourcesByName(
        new String[]{"SALES_AMOUNT", "PRODUCTS_DIM", "CUSTOMERS_DIM",
                     "CHANNELS_DIM", "TIMES_DIM", "PROMOTIONS_DIM"});
  Source salesAmount = sources[0];
  StringSource product = (StringSource)sources[1];
  StringSource customer = (StringSource)sources[2];
  StringSource channel = (StringSource)sources[3];
  StringSource time = (StringSource)sources[4];
  StringSource promo = (StringSource)sources[5];
  /*
   * Create a SingleSelectionTemplate to produce a Source that
```

```
     * specifies a single value for each of the dimensions other
     * than the base for the selected measure.
     */
    SingleSelectionTemplate singleSelections =
                        new SingleSelectionTemplate(salesAmount, dp);
    singleSelections.addSelection((StringSource) customer,
                                              "San Francisco");
    singleSelections.addSelection((StringSource) time, "2000-Q1");
    // S is the direct sales channel
    singleSelections.addSelection((StringSource) channel, "S");
    singleSelections.addSelection((StringSource) promo, "billboard");
    /*
     * Create a TopBottomTemplate and set the parameters selected by
     * the end user, including a dimension as the base and the
     * Source produced by the SingleSelectionTemplate as the
     * criterion.
     */
    TopBottomTemplate topNBottom = new TopBottomTemplate(product, dp);
    topNBottom.setTopBottomType(TopBottomTemplate.TOP_BOTTOM_TYPE_TOP);
    topNBottom.setN(15);
    topNBottom.setCriterion(singleSelections.getSource());
    /*
     * With methods on the TransactionProvider, prepare and commit
     * the transaction.
     */
    try{
      context.getTransactionProvider().prepareCurrentTransaction();
    }
    catch(NotCommittableException e){
      System.out.println("Cannot prepare current Transaction. " +
                         "Caught exception " + e + ".");
    }
    context.getTransactionProvider().commitCurrentTransaction();
    /*
     * Get the Source produced by the TopBottomTemplate,
     * create a Cursor for it and display the results.
     */
    createCursor(topNBottom.getSource(), dp);
  }
}
```

A
Setting Up the Development Environment

Appendix summary

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

List of topics

This chapter includes the following topics:

- Component Overview
- Location of Files on the OLAP Services Computer
- Setting Up on Your Application Development Computer
- Considerations for Deploying Your Application

Component Overview

Components provided by the Oracle installation

Chapter 1 describes the software components that are involved in an application that uses the OLAP API. The Oracle installation provides all of the components that run on the OLAP Services computer. In addition, the installation provides `jar` files that are needed on the application development computer for creating an OLAP API client application.

As an application developer, you must copy these `jar` files from the OLAP Services computer to the computer on which you will write your Java application. The files that you copy supply the OLAP API client software and CORBA software that is used by the OLAP API.

Components required on the application development computer

The application development computer must have the following components:

- OLAP API `jar` files, which represent the OLAP API client software. The Oracle installation provides these files, along with OLAP API reference documentation and sample Java progams that use the OLAP API.
- VisiBroker for Java `jar` files, which support the underlying CORBA connection between your application and OLAP Services. The Oracle installation provides these files.
- Oracle CORBA naming service `jar` files, which provide CORBA naming services so your application can find an OLAP service. The Oracle installation provides these files.
- The Java Development Kit (JDK) version 1.1.8. The Oracle installation does not provide the JDK. For information about obtaining and using it, see the Sun Microsystems Java Web site at `java.sun.com`.

Location of Files on the OLAP Services Computer

OLAP API jar files

The Oracle installation of OLAP Services places the following OLAP API `jar` files on the OLAP Services computer. It places them in the `olap/olapi/lib` subdirectory of the Oracle home directory.

```
collections.jar
express_common.jar
express_connection.jar
express_mdm.jar
express_olapi_common.jar
express_olapi_data.jar
express_olapi_data_full.jar
express_olapi_data_receiveOnly.jar
express_olapi_indep.jar
express_spl.jar
```

Note that the `collections.jar` file includes the Sun Microsystems backport of the collections framework from Java version 1.2.

VisiBroker for Java jar files

The installation places the following version 3.4 VisiBroker for Java `jar` files on the OLAP Services computer. It places them in the `lib` subdirectory of the Oracle home directory.

```
vbjorb.jar
vbjapp.jar
vbjtools.jar
```

Oracle CORBA naming service jar files

The installation places the following Oracle CORBA naming service `jar` files on the OLAP Services computer. It places them in the `lib` or `javavm/lib` subdirectory of the Oracle home directory.

```
aurora_client.jar
aurora_client_orbdep.jar
aurora_client_orbindep.jar
```

Setting Up on Your Application Development Computer

Installing the jar files

To make the `jar` files accessible in your development environment, perform the following steps:

1. Copy the OLAP API, VisiBroker, and Oracle CORBA naming service `jar` files to your application development computer. Place them in a location that makes them accessible to the Java integrated development environment (IDE) that you are using. An example of an IDE is Oracle JDeveloper.
2. Edit your Java `CLASSPATH` environment variable to include the paths of the files on your computer.
3. In the IDE, make any specifications that are required to make the files accessible for importing classes into your programs.

Installing the OLAP API reference documentation

If you want to access the *Oracle9i OLAP Services OLAP API Reference* files on your application development computer, locate the `jar` files that contain them in the `olap/olapi/doc` subdirectory of the Oracle home directory on the OLAP Services computer. Consult the `readme.txt` file in that directory for instructions on how to install the files and access them in your Web browser.

The *Oracle9i OLAP Services OLAP API Reference* was created using the javadoc tool provided by Sun Microsystems.

Installing the sample programs

If you want to run the Java sample programs on your application development computer, locate the jar file that contains them in the olap/olapi/shprog subdirectory of the Oracle home directory on the OLAP Services computer. Consult the readme.txt file in that directory for instructions on how to install the files and run the sample programs.

The sample programs access data and metadata that is in the Sales History schema, which is provided with the Oracle installation.

Considerations for Deploying Your Application

OLAP API and VisiBroker for Java classes

When you deploy an application, ensure that its configuration includes the OLAP API and VisiBroker for Java `jar` files.

CORBA naming service

If your application uses the Oracle CORBA naming service, ensure that you include the Oracle CORBA naming service `jar` files when you deploy your application. If you write your application for use with the VisiBroker Smart Agent naming service, make the software for that naming service available with your application.

Java Runtime Environment (JRE)

When you deploy your application, ensure that users are running a version of the Java Runtime Environment (JRE) that is compatible with the JDK version that you used for developing your application.

B

Using the Smart Agent Naming Service

Appendix summary

This appendix describes how to use the VisiBroker Smart Agent naming service in order to get a CORBA stub for making an OLAP API connection.

List of topics

This chapter includes the following topics:

- Role of a Naming Service in the Connection Process
- Getting the CORBA Stub Using VisiBroker Smart Agent

Role of a Naming Service in the Connection Process

Overview of connection steps

The connection process involves the following three steps:

1. Get a CORBA stub.
2. Create a `Properties` object for passing parameters to the `connect` method.
3. Call the `connect` method.

See Chapter 3 for a description of these steps.

How a naming service is used in the connection steps

A naming service locates an object, such as an OLAP service, that is accessible in the CORBA environment. A naming service is involved only in the first of the three connection steps, when you are getting a CORBA stub.

Different naming services on different platforms

Chapter 3 describes the code for getting a CORBA stub using the Oracle CORBA naming service. However, on some platforms, you should write code that uses the VisiBroker Smart Agent naming service instead. The rest of this appendix describes the code for getting the CORBA stub using VisiBroker Smart Agent.

See the read me file for your installation of OLAP Services for information about whether the procedures described in this appendix are relevant on your platform.

Getting the CORBA Stub Using VisiBroker Smart Agent

What is a CORBA stub?

A CORBA stub is a Java object that resides on the application computer and represents the OLAP service to which a connection will be made. To obtain the stub, you execute methods that are provided by a CORBA naming service.

What do you do with a CORBA stub?

In the connection steps that are described in Chapter 3, you pass the stub as a parameter to the `connect` method on a `ConnectionManager`. The stub helps to identify the service to which the connection will be made.

Code for getting the CORBA stub

The following sample code for getting the stub uses the Smart Agent naming service that is compatible with version 3.4 of VisiBroker for Java. The code initializes an `ORB` object (called myORB in the code) and creates a stub that represents an OLAP service. The code specifies the following three identifiers:

- The name of the computer on which the Smart Agent and the OLAP service reside. This host name was specified by a database administrator in the OSAgentAddr setting in OLAP Services Instance Manager. In this sample code, the name is LAB1.
- The port number for the Smart Agent. This name was specified by a database administrator in the OSAgentPort setting in OLAP Services Instance Manager. In this sample code, the port number is 10160.
- The name of the OLAP service. In this sample code, the service name is OLAPSrv1.

You can find out the computer name, the port number, and the OLAP service name by asking your OLAP Services database administrator.

```
import org.omg.CORBA.ORB;

Properties orbParams = System.getProperties();
String addrName = "ORBagentAddr";
String addrValue = "LAB1";
String portName = "ORBagentPort";
String portValue = "10160";
orbParams.put(addrName, addrValue);
orbParams.put(portName, portValue);
String[] dummyArgs = {"A", "B"};

ORB myORB = ORB.init(dummyArgs, orbParams);

String serviceString = "LAB1:OLAPSrv1";

org.omg.CORBA.Object serviceStub =
   ((com.visigenic.vbroker.orb.ORB) myORB).bind(
   "IDL:ExpressConnectionModule/ServerInterface:1.0",
   serviceString, null, null);
```

An alternative way to specify the Smart Agent computer name and port number is by providing them as parameters to the Java Runtime Environment (JRE) when the application executes. For our example, the following parameters could be specified when the application is executed. For each one, “-D” indicates that what follows is a JRE parameter.

```
-DORBagentAddr=LAB1 -DORBagentPort=10160
```

With these parameters specified, the application can use the `ORB.init()` method with no parameters.

Index