

SAP Business Intelligence Java Software Development Kit Developer's Guide



SAP NetWeaver™

Palo Alto, California

Copyright

© Copyright 2004-2006 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, OS/2, Parallel Sysplex, MVS/ESA, AIX, S/390, AS/400, OS/390, OS/400, iSeries, pSeries, xSeries, zSeries, z/OS, AFP, Intelligent Miner, WebSphere, Netfinity, Tivoli, and Informix are trademarks or registered trademarks of IBM Corporation in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

MaxDB is a trademark of MySQL AB, Sweden.

SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

Typographical Conventions

To denote Java source code, package, interface, class, path, and file names, we use the `Courier New` font. If rendering and viewing this document with color capability, you will in addition see the following Java code coloring conventions assigned to example code snippets:

Code comments – teal:

```
// display HTML
```

Java keywords – **boldfaced violet**:

```
public void doGet(HttpServletRequest request,
```

Java strings – blue:

```
out.println("<html>");
```

Code excerpts are displayed in boxes.

Icons

Icons used in this Guide may include the following:

Icon	Meaning
	Caution
	Example
	Note
	Recommendation or tip

Table of Contents

Introduction.....	1
Chapter 1: Getting Started	12
Chapter 2: Connecting to Data Sources.....	31
Chapter 3: Accessing Metadata	42
Chapter 4: Creating Queries	64
Chapter 5: Retrieving Result Sets	106
Chapter 6: Exceptions	124
Appendix A: Installation	A-1
Appendix B: Examples.....	B-1
Appendix C: Additional Resources.....	C-1
Appendix D: Glossary	D-1
Index.....	1

Detailed Table of Contents

Introduction.....	1
Overview of the BI Java SDK.....	1
Components of the SDK	2
Application Programming Interfaces	2
Documentation	3
Examples.....	4
BI Java Connectors.....	4
Architectural Overview	5
Open Standards in the SDK	5
Foundation Technologies Versions	11
Chapter 1: Getting Started	12
Overview	12
System Configuration	12
Hello MDX: First Example for a Multidimensional Data Source.....	14
Step 1: Import Packages	14
Step 2: Connect to an SAP BI System.....	15
Step 3: Retrieve the Metadata	18
Step 4: Create a Query.....	18
Step 5: Change the Layout.....	19
Step 6: Specify Selected Members.....	20
Step 7: Execute the Query and Retrieve the Result Set.....	20
Step 8: Render the Result Set.....	20
Hello SQL: First Example for a Relational Data Source	23
Step 1: Import Packages	24
Step 2: Connect to a JDBC Database	24
Step 3: Retrieve the Metadata	27
Step 4: Create a Query.....	27
Step 5: Specify Table and Columns	28
Step 6: Execute the Query and Retrieve the Result Set.....	28
Step 7: Render the Result Set.....	29
Chapter 2: Connecting to Data Sources.....	31
Overview	31
Connection Architecture.....	32
Client Interface	33
Portal Connection Framework	34
Service Provider Interface	34
Managed Environment.....	34

Non-managed Environment.....	35
Connection Specification and Portal Service.....	36
BI Java Connectors.....	37
BI JDBC Connector.....	39
BI ODBO Connector.....	39
BI SAP Query Connector.....	40
BI XMLA Connector.....	40
Examples.....	40
Chapter 3: Accessing Metadata.....	42
Overview.....	42
Common Warehouse Metamodel.....	42
Generating Interfaces.....	43
Metadata APIs.....	45
OLAP Metadata Model.....	46
Relational Metadata Model.....	57
Examples.....	62
Chapter 4: Creating Queries.....	64
Overview.....	64
Query APIs.....	64
OLAP Query Model.....	65
Relational Query Model.....	95
Examples.....	101
OLAP Queries.....	101
Relational Queries.....	104
Chapter 5: Retrieving Result Sets.....	106
Overview.....	106
ResultSet API.....	106
Key Features.....	107
OLAP Result Sets.....	108
Relational Result Sets.....	115
OLAP Table Model.....	116
Examples.....	118
Chapter 6: Exceptions.....	124
Overview.....	124
Exception Handling.....	124
Exception Translation.....	125
Appendix A: Installation.....	A-1
Overview.....	A-1
System Requirements.....	A-1
Classpath Configuration.....	A-2

Logging and Tracing, JARM.....	A-2
Using the BI XMLA Connector in a non-managed environment.....	A-3
Documentation	A-4
How-To Guides.....	A-4
BI Java Connectors.....	A-5
BI Java Connectors Overview	A-6
Testing the BI Java Connectors.....	A-7
Appendix B: Examples.....	B-1
Overview	B-1
Finding the Examples.....	B-1
Configuring your System.....	B-2
Data Sources.....	B-2
Rendering to File	B-3
Connection Properties	B-3
Index of Examples.....	B-3
Appendix C: Additional Resources.....	C-1
Web References	C-1
Books	C-3
Appendix D: Glossary	D-1
Index.....	1

Introduction

This Guide helps developers use the SAP Business Intelligence Java Software Development Kit (BI Java SDK). With the BI Java SDK, you can build analytical applications that access, manipulate, and display both multidimensional (Online Analytical Processing, or OLAP) and tabular (relational) data. The target audience is Java developers with business intelligence experience and needs.

This chapter provides an overview of the BI Java SDK and the BI Java Connectors and introduces their foundation of open standards technologies.

Overview of the BI Java SDK

SAP Business Intelligence (BI – or earlier BW) front-end tools and their business content fulfill many business intelligence needs. Now, with the addition of the BI Java SDK, you can develop highly customized business analytics to complement your existing business intelligence solution, using the BI Java SDK as SAP's main interface for accessing any OLAP or relational data your applications or client components need.

SAP BI's existing Open Analysis Interfaces — XML for Analysis, the OLAP BAPI, and OLE DB for OLAP — allow you to access SAP BI or external OLAP data for your custom application needs. The BI Java SDK provides additional capabilities and simplifies the task of implementing client applications based on these interfaces. The SDK encapsulates underlying low-level communication protocols such as HTTP, and simplifies the generation of complex MDX or SQL query statements.

Although the primary intention of the SDK is to simplify programming against the BI OLAP engine within a Java environment, the application programming interfaces (APIs) can also be used to access non-SAP and even non-OLAP data sources, such as relational JDBC data sources. This allows programmers to work with a single uniform approach throughout development of an application.

The BI Java SDK provides an object-oriented framework in which to:

- Connect to a variety of data sources
- Access master- and metadata
- Create and execute complex queries
- Render and access query result sets

A driver-based architecture supports access to different data formats using a variety of protocols:

- Java Database Connectivity (JDBC)
- OLE DB for OLAP (ODBO)

Components of the SDK

- SAP Query
- XML for Analysis (XMLA)

Components of the SDK

The BI Java SDK consists of a set of application programming interfaces (APIs), documentation, and examples, described below.

Application Programming Interfaces

The following diagram provides a simplified picture of the APIs of the BI Java SDK:

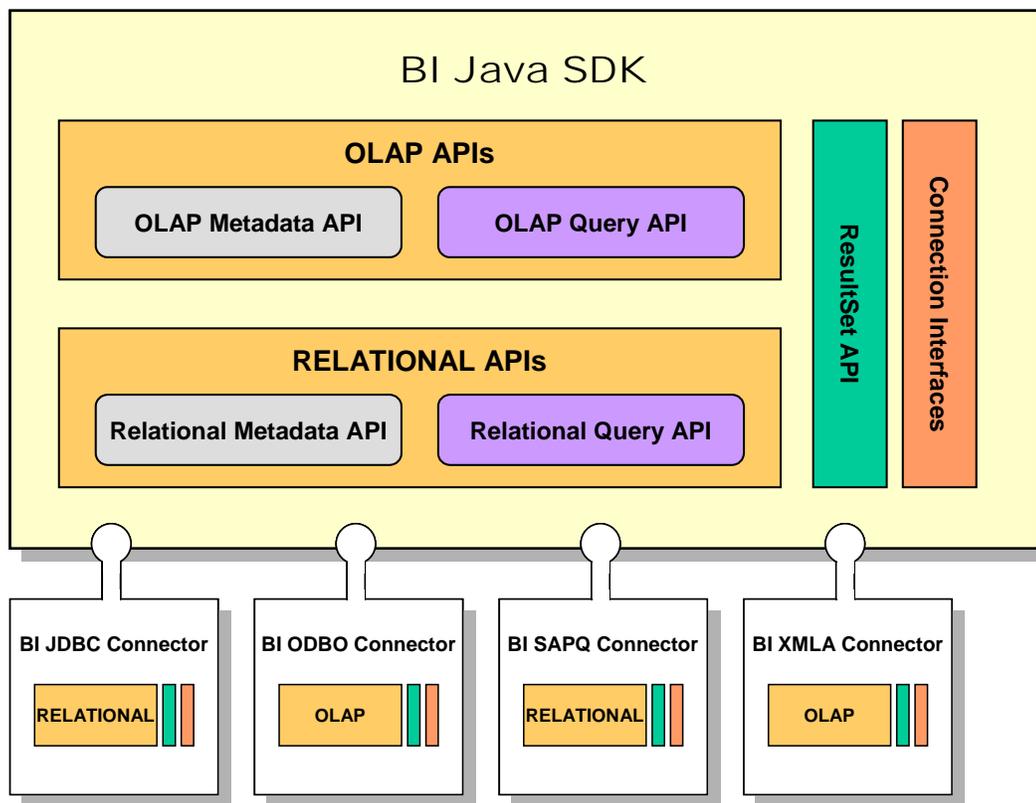


Figure 1 — BI Java SDK APIs

Components of the SDK

As pictured above, the BI Java SDK consists of the following APIs:

- OLAP APIs:
 - OLAP Metadata API, for accessing OLAP metadata
 - OLAP Query API, for creating, processing, and navigating queries against an OLAP data source
- Relational APIs:
 - Relational Metadata API, for accessing relational metadata
 - Relational Query API, for creating, processing, and navigating queries against a relational data source
- Common ResultSet API, for accessing and rendering either OLAP or relational result sets
- Connection Interfaces

For detailed architectural information about the SDK's APIs, refer to the following chapters of this Guide:

- Connection Interfaces: see [Connecting to Data Sources](#)
- Metadata APIs: see [Accessing Metadata](#)
- Query APIs: see [Creating Queries](#)
- ResultSet API: see [Retrieving Result Sets](#)

In addition, the SDK provides an exception framework, which you can read about in the [Exceptions](#) chapter.

Documentation

In addition to the APIs, the SDK includes comprehensive documentation, including this Developer's Guide with step-by-step tutorials (see [Getting Started](#)), Javadocs with package and overview documentation, and additional supporting HTML documents. Launch the package from the Javadoc [./index.html](#) file located in the docs folder of the unpackaged distribution archive.

The names of the interfaces exposed in the SDK's APIs begin with the prefix IBI. Many of the concepts are introduced in the documentation together with UML class diagrams to illustrate the links between the different Java interface components.

For complete interface descriptions, refer to the corresponding Javadocs, included in the SDK distribution package. We may include direct links to the Javadoc HTML pages for specific classes or packages, or you can navigate through the Javadocs yourself:

- [SDK Javadocs overview page](#)
- [CWM Javadocs overview page](#)

- [Documentation set root](#) – the Javadoc index page in the doc folder of the SDK distribution package

 **Caution:**

In many cases, this Guide provides links to other components of the documentation set, for example, to Javadoc packages, so that you can easily navigate to them. These links are set relative to the original location of the Guide, in the `docs/devguide` folder of the distribution package, and will not resolve if the `devguide.pdf` file is removed from this folder.

The links have been tested on Windows systems only, and may not work on other systems.

Examples

Examples provide easy-to-use Java servlets which demonstrate many aspects of our query APIs, as well as step-by-step connection instructions. Examples subsections at the end of each chapter, where relevant, show usage of the SDK's APIs in real-world scenarios. Although each example may embody all of the SDK's APIs, we introduce the examples within the context of the API most emphasized in the given example.

The examples are also cataloged in the documentation set (begin with the [index.html](#) file located in the docs folder of the unpackaged distribution archive and see the Examples link) and in Appendix B: Examples. Refer to these sections for the system configuration information you'll need to get up and running.

BI Java Connectors

The BI Java Connectors are a group of four JCA (J2EE Connector Architecture)-compliant resource adapters that implement the BI Java SDK's APIs and allow you to connect the applications you build with the SDK to heterogeneous data sources. They may be deployed onto SAP NetWeaver's J2EE Web Application Server.

The BI Java SDK contains the JAR files you need to develop applications using any of the BI Java Connectors and to use them in an unmanaged scenario, but to use your application with a data source in the managed environment of the J2EE server, you need to deploy the appropriate BI Java Connector. The connectors themselves are deployed with NetWeaver's optional BI UDI component.

Four connectors are available:

- BI JDBC Connector, for connecting to relational JDBC data sources
 - implements the SDK's relational APIs
- BI ODBO Connector, for connecting to OLE DB for OLAP-compliant data sources
 - implements the SDK's OLAP APIs
- BI SAP Query Connector, for connecting to data from SAP operational applications
 - implements the SDK's relational APIs

Architectural Overview

- BI XMLA Connector, for connecting to XMLA-compliant data providers such as SAP BI systems (BW)
 - implements the SDK's OLAP APIs

For more information about the connectors, see the BI Java Connectors section of this Guide.



Note:

The BI Java Connectors are distributed separately from the BI Java SDK, deployed with NetWeaver's optional BI UDI component.

Architectural Overview

The SDK's components are architected to simplify the integration of data from diverse data sources by displaying a unified metadata model and common access interface. We leverage the concepts of open standards such as the Java Metadata Interface (see Java Metadata Interface, below) and the Common Warehouse Metamodel (see Common Warehouse Metamodel, below) to support accessing and representing metadata. This approach hides the complexity and details of the underlying communication and access protocols (such as HTTP), enabling you to focus instead on your specific business requirements.

Another benefit of this approach is that applications are written only once for different types of data sources and operating systems. Developers do not need to understand the specifics of a particular system. Our driver-based architecture allows you to use a set of connectors, the BI Java Connectors, to connect applications to SAP data sources such as the SAP Business Information Warehouse, as well as to non-SAP data sources such as relational JDBC-based databases.

An additional key architectural feature of the SDK is the use of OLAP and Relational Query Models. These models provide interfaces for defining complex multidimensional or relational queries without negotiating the details and complexity of the specific query language—for example, MDX in the case of OLAP data sources. You need only interact with a simplified command API.

The rest of this section discusses the importance of open standards in the SDK and summarizes key individual standards. The SDK's architecture is discussed in more detail in the subsequent chapters of this Guide.

Open Standards in the SDK

Instead of designing a new API entirely from scratch, the SDK development team began by capitalizing upon the best of available open standards and technologies.

Architectural Overview

This approach has several advantages. Open standards provide high-quality specifications that have undergone a significant review process and are based on the experience of many developers. Reliance upon open standards also increases the chances for interoperability between components from different vendors, helps toward building easy-to-learn APIs, and supports potential synergies with other emerging J2EE standards. There are currently more than 100 Java technology specifications under development in the Java Community Process (JCP) program, including the next versions of Java 2 Micro Edition (J2ME), Java 2 Platform Enterprise Edition (J2EE), and Java 2 Standard Edition (J2SE). Many of these specifications are related or interconnected to the point that there are many synergies among them.

The following open platforms and standards are of particular relevance to the SDK architecture:

- The J2EE Platform
- J2EE Connector Architecture (JCA)
- Common Warehouse Metamodel (CWM)
- Java Metadata Interface (JMI)
- Java Database Connectivity (JDBC)
- Meta Object Facility (MOF)
- XML Metadata Interchange (XMI)
- OLE DB for OLAP (ODBO) and XML for Analysis (XMLA)

The following sections describe these standards and explain the rationale for using them in the SDK.

The J2EE Platform and Java Development with SAP NetWeaver

By leveraging the benefits of Sun's J2EE platform, the BI Java SDK provides an object-oriented framework for building multi-tier, Web-centric analytical applications that are:

- Portable
- Highly scalable
- Secure
- Reusable component-based

SAP's NetWeaver combines Java technologies with proven SAP programming models and technologies. With the Web Application Server, SAP offers a runtime environment for Web applications that can be written both in ABAP and in Java. A fully J2EE-compliant server has been integrated with the traditional SAP Web Application Server, providing fast connectivity between applications written in Java and ABAP. The SAP Web Application Server, therefore, has two personalities that allow us to write purely Java-based applications:

- The straightforward J2EE programming model

Architectural Overview

- The ABAP/Java mixture: a combination of existing functionality written in ABAP with new components developed in Java

The BI Java Connectors, used by the SDK to connect to diverse data sources, are optimized for full compliance and deployment to SAP NetWeaver's Web Application Server, which provides additional enhancements to the following areas of J2EE-standard compliance:

- Stability
- Serviceability
- Performance and Scalability
- Improved Administration
- Replacement of the internal database by open JDBC.

J2EE Connector Architecture

Sun's J2EE Connector Architecture (JCA) defines a standard architecture for connecting the Java 2 Platform to heterogeneous Enterprise Information Systems (EISs) such as mainframe transaction processing and database systems. The connector architecture satisfies the SDK's needs for a pluggable driver-based architecture, required for achieving the openness necessary to integrate with a variety of data sources.

The SAP J2EE engine is JCA compliant. SAP provides a Java Connector (JCo) that allows Java applications to communicate with any SAP R/3 system. The Portal Runtime offers additional services on top of JCA for persisting connection information in a SystemLandscape, and a ConnectorGateway service that integrates with the portal user management.

The SDK leverages JCA to provide consistent connection management to heterogeneous data sources and to support the pluggability of connectors into multiple J2EE engines.

Advantages to the SDK of JCA compliance include:

- **Support for connectivity to heterogeneous data sources:**
The SDK provides connectivity via data source-specific connectors that conform to the JCA patterns. This is a modular and pluggable architecture, where new adapters can be easily added. The BI Java Connectors provide a BI-specific tailored connection that serves as the entry point to any services.

The BI Java Connectors are kept lightweight, so that connectors deal only with the specific domain of the resource to which they connect. For example, the BI JDBC Connector has only to deal with mapping relational metadata into the CWM Relational-based JMI service. Any type of metadata (queries, OLAP, and relational) is represented via JMI-compliant interfaces and implementations that are generated by the JMI mapping service of SAP's Metamodel Repository.

- **Leveraging of SAP's NetWeaver Portal services:**
The BI Java Connectors integrate with the Portal runtime by extending the Generic Connector interfaces. Thus,

the Java components can leverage services provided by the Portal such as `SystemLandscape` and `ConnectorGateway`.

Common Warehouse Metamodel

The Common Warehouse Metamodel (CWM) is an Object Management Group (OMG) standard that provides a framework for representing metadata in data warehousing, business intelligence, knowledge management, and portal technologies.

CWM is of particular relevance to the BI Java SDK because BI applications are typically strongly driven by metadata. We chose CWM as the common metamodel for the SDK, grounding our architecture in a solid, standardized metadata approach. CWM is currently gaining momentum in the industry, and provides the required expressiveness to model metadata from different implementations of OLAP and relational models. The SDK's OLAP and Relational Metadata Models are provided entirely by CWM.

The SDK leverages CWM metamodels for various additional reasons:

- CWM conforms to the MOF standard, which allows the SDK to apply JMI mappings to render a standard API for manipulating and navigating instances of the model.
- CWM metamodels are capable of modeling a wide spectrum of OLAP and relational providers. They are not only generic and extensible in their overall content and structure, but are also separated from implementation considerations. This is of particular importance for the SDK, where a large variety of providers are mapped into one common metamodel.
- CWM provides a complete relational metamodel that is based on the SQL standard. Therefore, defining a relational metadata service for accessing tabular data providers like JDBC is straightforward.
- CWM provides an OLAP metamodel that contains the essential OLAP concepts common to most OLAP systems.

Java Metadata Interface

The OMG's Java Metadata Interface (JMI) specification defines a platform-neutral infrastructure that enables the creation, storage, access, discovery, and exchange of metadata. JMI defines a Java mapping for the MOF. It can be viewed as an extensible metadata service for the Java platform that provides a common Java programming model for accessing metadata. Any system that provides a JMI-compliant API to its public metadata is a JMI service. JMI provides the following to the J2EE environment:

- A metadata framework that provides a common Java programming model for accessing metadata.
- An framework for integration and interoperability for Java tools and applications.
- Integration with OMG modeling and metadata architecture.

Architectural Overview

As the Java rendition of the MOF, the JMI specifies a set of rules that generate, for any given MOF-compliant metamodel, a set of Java APIs for manipulating the information contained in the instances of that metamodel.

The JMI specification also contains a Java implementation of MOF reflection. Although reflective capabilities are more relevant for advanced tools, they are of interest for the SDK, because by supporting the reflective interfaces we automatically gain support for XMI.

In the SDK, JMI is used to render the CWM model into programmatic APIs. It is specifically geared for a Java rendering of MOF-compliant metamodels and its mapping templates provide a uniform and flexible Java API for manipulating and accessing data based on the CWM models.

Java Database Connectivity

Java Database Connectivity (JDBC) is Sun's Java API that provides access to virtually any relational data source from within the Java programming language. It provides cross-DBMS connectivity to a wide range of SQL databases, and also provides access to other tabular data sources such as spreadsheets or flat files. Sun maintains a database of JDBC-enabled drivers currently containing more than 170 entries, which indicates the broad industry support for this API.

In the SDK, JDBC is respected as the existing Java standard API for accessing result sets. JDBC is widely used, and many Java developers are familiar with the ResultSet API. We only needed to add a few extensions to utilize the ResultSet API with multidimensional (OLAP) datasets as well as relational result sets, and in this way the SDK's ResultSet API achieves a common look-and-feel across relational or OLAP result sets.

Meta Object Facility

The MOF is an OMG standard which provides an open-ended information modeling capability. MOF consists of a base set of meta-modeling constructs used to describe technologies and application domains, and a mapping of those constructs to CORBA IDL (Interface Definition Language) for automatically generating model-specific APIs. The MOF also defines a reflective programming capability that allows applications to query a model at runtime to determine the structure and semantics of the modeled system.

Although MOF is not directly utilized or exposed by the SDK, it is important as a foundation technology, and many of the SDK interfaces are the result of mapping MOF-compliant metamodels using the JMI code templates.

XML Metadata Interchange

XML Metadata Interchange (XMI) is an OMG standard that supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information. XMI integrates the Unified Modeling Language (UML), MOF, and XML, and allows developers of distributed systems to share object models and other metadata over the Internet. XMI, together with MOF and UML, form the core of the OMG metadata repository architecture. There are many advantages of basing a metadata interchange format on XML. These include the following:

- XML is an open, platform-independent, and vendor-independent standard.
- XML supports the international character set standards of extended ISO Unicode.
- XML is metamodel-neutral, and can represent metamodels compliant with OMG's meta-metamodel, the MOF.
- In the SDK, implementing the metadata services exposed by the connectors as full-blown JMI services automatically provides XMI support. A JMI service provides APIs for streaming metadata in the XMI format. The `XmiWriter` and `XmiReader` interfaces import and export XML documents to and from a JMI service.

XMI, together with MOF, is important in the SDK as a foundation technology. By applying a JMI rendering to interfaces, we also provide XMI capabilities, which support a common exchange format for metadata.

OLE DB for OLAP and XML for Analysis

Microsoft's OLE DB for OLAP is the de-facto industry standard API for exchanging metadata and data between an OLAP server and a client on a Windows platform. (Throughout this Guide, we abbreviate OLE DB for OLAP with "ODBO.") Microsoft's XML for Analysis (XMLA) is advancing this standard by leveraging many of the established concepts of ODBO for a Web services API. Both ODBO and XMLA utilize a SQL-like query language called MDX (Multidimensional Expressions). Since MDX-based OLAP providers are currently the de-facto standard, the SDK takes capabilities of such providers into account.

Although many of the expressions used in the SDK's OLAP Query API are common to different OLAP implementations, they were designed with MDX-based providers and mind and therefore resemble various expressions described in the MDX grammar in some areas. Note however that the SDK takes a strongly object-oriented approach to defining queries, rather than a linguistic approach.

Foundation Technologies Versions

The table below lists the version of each foundation technology used in the BI Java SDK:

Foundation Technology	Supported/Required version
SAP NetWeaver '04 and greater	Web Application Server 6.40 and greater
Business Information Warehouse (SAP BI Systems – or BW)	BW 3.5 and greater
Meta Object Facility (MOF)	1.4
Java Metadata Interface (JMI)	1.0
Common Warehouse Metamodel (CWM)	1.0
J2EE Connector Architecture (JCA)	1.0
Java Database Connectivity (JDBC)	2.0
Java Development Kit (JDK)	1.4
XML Metadata Interchange (XMI)	1.2

Chapter 1: Getting Started

Overview

This chapter contains two complete tutorials to jump-start you in using the BI Java SDK in end-to-end scenarios, from establishing a connection to rendering the results into an HTML page:

- Hello MDX
- Hello SQL

We excerpt from the code below to illustrate and comment on the major steps. The complete source code of these examples, from the SDK package `com.sap.ip.bi.sdk.samples`, can be found in `Tutorial_1.java` (MDX) and `Tutorial_2.java` (SQL) in the `docs/examples` folder after unzipping the BI Java SDK. There, you can also view the HTML rendering of the result.

System Configuration

First, we introduce the technical requirements and configuration instructions for running these tutorials.

Data Source Requirements

Hello MDX

This chapter's OLAP tutorial, "Hello MDX," uses the BI XMLA Connector JAR file included with the SDK and is based on the SAP BI demo content InfoCube `OD_SD_C03` "SAP Demo: Sales and Distribution Overview" and the query `OD_SD_C03/OD_SD_C03_Q009` "Order and Sales values." To reproduce this tutorial, you need access to SAP BI system release 2.0 or higher with this SAP demo content activated.

This tutorial runs locally in an unmanaged environment, so you do not need the BI XMLA Connector itself to reproduce it.

Hello SQL

This chapter's SQL tutorial, "Hello SQL," uses the BI JDBC Connector JAR file included with the SDK and a JDBC data source. To reproduce this tutorial, you need an active JDBC data source for which you have a valid user name and password, and you need to have properly configured your JDBC driver in your local environment.

In contrast to the OLAP tutorial, this tutorial doesn't depend on specific data in a specific data source. We will show you how to retrieve the data in all the columns of the first table in your JDBC database, so results on different databases will vary.



Caution:

Note that the relational tutorial's sample class, `Tutorial_2.java`, depends on the presence of your database provider's JDBC driver in your classpath. If this is not configured correctly, this tutorial will fail.

Connection Properties

We rely on easily accessed and edited properties files to supply your connection-specific information such as username and password to the connectors. We include four properties files, one for each BI Java Connector, in the examples folder nested in the `com.sap.ip.bi.sdk.samples` package with the full set of Java source files. For this tutorial, you will want to work with the XMLA or the JDBC properties file:

- [Helpers.xmla.properties](#): connection information for the BI XMLA Connector, used in "Hello MDX" Tutorial 1.
- [Helpers.jdbc.properties](#): connection information for the BI JDBC Connector, used in "Hello SQL" Tutorial 2.

Edit the existing files with your own connection information, or create new files in the `com.sap.ip.bi.sdk.samples` package to locally override the properties, named `Helpers.nnnn.local.properties` (where "nnnn" corresponds to the four-letter connector name). The `Helpers` class will first look for the local file, and if not found, will take the original properties files.

For connector properties configuration information, refer to the `howto.html` file that ships inside of each resource adapter archive or is available in the SDK documentation set, on the Connectors page.



Note:

The connectors' `howto.html` files are also included in the SDK distribution package for your reference. See [index.html](#) in the docs folder of the package, then follow the Connectors link.

Rendering the Servlets

These tutorials, like all the SDK examples, implement a minimal HTTP servlet, which generates HTML for easy viewing of results. By default, running the main method without a parameter will write the HTML to the console. To write it to an HTML file instead for easy viewing in a Web browser, specify a filename, with full path and `.html` extension, as the parameter.

Hello MDX: First Example for a Multidimensional Data Source

Our OLAP version of “Hello World” demonstrates the complete code sequence required to display a result on the screen. In this tutorial, you build a servlet that connects to an OLAP data source, accesses its metadata, builds a query, executes it, and accesses and renders the result set. You will actually be defining and rendering the result of an MDX query, but with our OLAP Query API, the complexity of the underlying MDX statement is hidden.

The figure below shows a basic result set created with a query in the BEx Analyzer when drilling down by sales organization (OD_SALE_ORG) with the key figures on the columns. This view also has a single filter value on division set to “High Tech”:

Sales organization	Incoming orders value	Billing value	Returns value	Credit memo value
Frankfurt	~7,551,870.83~ EUR	~7,278,661.82~ EUR	~205,535.69~ EUR	~169,745.46~ EUR
Berlin	~11,445,868.36~ EUR	~11,215,095.32~ EUR	~291,228.97~ EUR	~218,216.68~ EUR
Munich	~22,297,907.43~ EUR	~20,821,494.95~ EUR	~1,227,705.02~ EUR	~1,213,388.94~ EUR
Hamburg	~6,388,127.30~ EUR	~6,205,290.09~ EUR	~492,882.69~ EUR	~490,019.07~ EUR
London	£ ~14,984,849.00~	£ ~14,569,213.00~	£ ~679,660.00~	£ ~354,560.00~
Birmingham	£ ~9,504,655.00~	£ ~9,272,423.00~	£ ~423,060.00~	£ ~207,440.00~
Paris	~29,869,563.66~ EUR	~28,069,450.11~ EUR	~2,513,570.66~ EUR	~1,536,671.01~ EUR
New York	\$ ~24,722,543.00~	\$ ~24,109,951.00~	\$ ~1,574,226.00~	\$ ~1,547,560.00~
San Francisco	\$ ~8,550,079.00~	\$ ~7,917,756.00~	\$ ~718,226.00~	\$ ~718,226.00~
Toronto	~31,519,421.00~ CAD	~30,570,831.00~ CAD	~2,497,600.00~ CAD	~1,364,400.00~ CAD
Vancouver	~25,598,113.00~ CAD	~25,166,071.00~ CAD	~1,218,110.00~ CAD	~704,710.00~ CAD
Sydney	~1,741,800.00~ AUD	~1,649,767.00~ AUD	~57,263.00~ AUD	~85,895.00~ AUD
Overall Result	*	*	*	*

Figure 2 — Example BEx Result Set

The goal of this tutorial is to use the BI Java SDK to create a servlet with a result similar to this.

Step 1: Import Packages

First, configure the imports required for this SDK servlet. You may import by package, but we recommend you import by specific interfaces. For this tutorial, you will need the basic data access, connectivity, query generation, and supporting interfaces. The full list of imports is described below:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Locale;
import java.util.Properties;

import javax.resource.spi.ManagedConnectionFactory;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.omg.cwm.analysis.olap.Cube;
import org.omg.cwm.analysis.olap.Dimension;

import com.sap.exception.IBaseException;
import com.sap.ip.bi.sdk.dac.connector.IBIConnection;
import com.sap.ip.bi.sdk.dac.connector.IBIOlap;
import com.sap.ip.bi.sdk.dac.connector.olap.odbo.BIOdboMetadataConstants;
import com.sap.ip.bi.sdk.dac.connector.xmla.XmlaConnectionConstants;
import com.sap.ip.bi.sdk.dac.connector.xmla.XmlaManagedConnectionFactory;
import com.sap.ip.bi.sdk.dac.olap.query.IBICommandProcessor;
import com.sap.ip.bi.sdk.dac.olap.query.IBIMemberFactory;
import com.sap.ip.bi.sdk.dac.olap.query.main.IBIQuery;
import com.sap.ip.bi.sdk.dac.olap.query.member.IBIMember;
import com.sap.ip.bi.sdk.dac.result.IBIDataSet;
import com.sap.ip.bi.sdk.dac.result.model.BIDataSetTableModel;
import com.sap.ip.bi.sdk.dac.result.model.BITableItem;
import com.sap.ip.bi.sdk.exception.BIException;
import com.sap.ip.bi.sdk.exception.BISQLException;
import com.sap.ip.bi.sdk.localization.sdk.samples.Samples;
import com.sap.ip.bi.sdk.samples.servlet.MinimalServletContainer;
import com.sap.ip.bi.sdk.util.impl.BIResourceProperties;
import com.sapportals.connector.connection.IConnectionFactory;
import com.sapportals.connector.connection.IConnectionSpec;
```

Step 2: Connect to an SAP BI System

The next step is to connect to the SAP BI system. A connection to a data source is represented by an instance of the `IBIConnection` interface.

First create a basic servlet and prepare to read the connection information from the properties helper file, with help from `Helpers.java`:

Hello MDX: First Example for a Multidimensional Data Source

```
public class Tutorial_1 extends HttpServlet {
    private final static String CONTENT_TYPE = "text/html";

    private static Properties connProp =
        new BIResourceProperties(Helpers.class, ".xmla");

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doGet(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType(CONTENT_TYPE);

        PrintWriter out = response.getWriter();
```

To establish a connection, you must provide connection information in your `Helpers.xml.properties` file that encapsulates all relevant parameters for a specific system. Because you are connecting to an SAP BI server, the server name, system number, client, user name, password, and language parameters are required.



Note: How to find the URL of your XMLA provider

For an SAP BI system, you can find this information by executing the function module `RSBB_URL_PREFIX_GET` under transaction `SE37`. For the import parameters of the function module, use the following values:

`I_HANDLERCLASS = CL_RSR_MDX_SOAP_HANDLER`

`I_PROTOCOL = HTTP`

`I_MESSAGESERVER =`

The URL path is always `*/sap/bw/xml/soap/xmla`.

Get the connection properties with which to create the connection from the properties file with the following lines:

```
try {
    ManagedConnectionFactory mcf;
    IConnectionFactory cf;
    IConnectionSpec cs;

    mcf = new XmlaManagedConnectionFactory();
    cf = (IConnectionFactory) mcf.createConnectionFactory();
    cs = cf.getConnectionSpec();

    cs.setPropertyValue(
        XmlaConnectionConstants.USERNAME.toString(),
        connProp.getProperty(XmlaConnectionConstants.USERNAME.toString()));

    cs.setPropertyValue(
        XmlaConnectionConstants.PASSWORD.toString(),
        connProp.getProperty(XmlaConnectionConstants.PASSWORD.toString()));

    cs.setPropertyValue(
        XmlaConnectionConstants.URL.toString(),
        connProp.getProperty(XmlaConnectionConstants.URL.toString()));

    cs.setPropertyValue(
        XmlaConnectionConstants.DATA_SOURCE.toString(),
        connProp.getProperty(XmlaConnectionConstants.DATA_SOURCE.toString()));

    cs.setPropertyValue(
        XmlaConnectionConstants.STATEFULNESS.toString(),
        connProp.getProperty(XmlaConnectionConstants.STATEFULNESS.toString()));

    cs.setPropertyValue(
        XmlaConnectionConstants.LANGUAGE.toString(),
        connProp.getProperty(XmlaConnectionConstants.LANGUAGE.toString()));
}
```



Note:

Be sure to prepare the `Helpers.xmla.properties` file with your own connection parameters.

Actually create the connection now by instantiating `IBIConnection`. In establishing the connection to an OLAP data source, the `IBIOlap` interface provides an entry point to access metadata and execute queries:

```
IBIConnection connection = (IBIConnection) cf.getConnectionEx(cs);
IBIOlap olap = connection.getOlap();
```



Note:

For more information on the SDK's connection architecture, see the `Connecting to Data Sources` chapter of this Guide.

Step 3: Retrieve the Metadata

Standard client applications need access to metadata to construct queries. Now that you're connected to your data source, you will retrieve its metadata. In this example, you get an interface to a specific cube of the SAP BI system to which you are connected. BEx queries are mapped to cubes.

For this example you will need the "0D_SD_C03/0D_SD_C03_Q009" cube and the two dimensions "0D_SALE_ORG" and "0D_DIV". You retrieve the metadata using the `ObjectFinder` provided by `IBIOlap`. The code sample below also contains a sanity check to ensure that the required cube is available:

```
Cube cube = olap.getObjectFinder().
    findCubeFirst((String)null, "0D_SD_C03/0D_SD_C03_Q009");
if (cube==null) {
    throw new BIException(Locale.getDefault(),
        Samples.SDK_SAMPLES_1000,
        new Object[] {"0D_SD_C03/0D_SD_C03_Q009"});
}
Dimension salesOrgDimension = olap.getObjectFinder().
    findDimensionFirst(cube, "0D_SALE_ORG");
Dimension divisionDimension = olap.getObjectFinder().
    findDimensionFirst(cube, "0D_DIV");
```

To filter the data by the "[0D_DIV] . [7]" division, create the corresponding member:

```
IBIMemberFactory queryFactory = olap.getQueryFactory().getMemberFactory();
HashMap taggedValues = new HashMap();
taggedValues.put(BIOdboMetadataConstants.MEMBER_UNIQUE_NAME,
    "[0D_DIV] . [7]");
IBIMember divisionHiTechMember = queryFactory.
    createMember(divisionDimension, "7", null, taggedValues);
```



Note:

For more information on the SDK's metadata APIs, see the [Accessing Metadata](#) chapter of this Guide.

Step 4: Create a Query

You're now ready to create a query. Query construction is done by `olap.createQuery(cube)`, and `IBIQuery` is the outcome.

Hello MDX: First Example for a Multidimensional Data Source

Querying is greatly assisted by the use of the OLAP Command Processor. The OLAP Command Processor is part of the OLAP Query API and makes it easier to use the underlying query model by hiding the complexity of this model. With it, you can create and manipulate complex queries with simple commands. You can think of the individual methods of the command processor in terms of macros that consist of several method calls manipulating the structures of queries.

Next, you therefore create a query and an instance of the OLAP Command Processor associated with this query:

```
IBIQuery query = olap.createQuery(cube);  
IBICommandProcessor commandProcessor = query.getCommandProcessor();
```

In its initial state, this query has the measures dimension of the associated cube drilled down on the columns axis and all other dimensions on the slicer axis, and will select all members of their respective dimensions.

Although a query can be executed immediately after its creation, executing it would not be very useful because the query does not contain specific information. Because nothing is selected for the dimensions by default, all dimensions are on the slicer axis, with default values that are mostly ALL members. Such a query would return a single cell on the columns axis for all available members. You therefore need to modify the query by changing its layout and specifying members.



Note:

For more information on the SDK's query APIs, see the Creating Queries chapter of this Guide.

Step 5: Change the Layout

“Layout” refers to the distribution of the dimensions and the axes of a query. In a two-dimensional layout, dimensions can be oriented along the columns or rows axis, and the selected members of the dimensions are visible on these axes of the result set. The slicer axis is used to filter the query by dimensions, which are oriented along this axis.

The OLAP Command Processor's `moveDimensionTo[Axis].(dimensionUniqueName)` methods append a dimension to the last position on the target axis. Additional methods provide more control with the positioning, which is important if there is already a dimension on the target axis.

To achieve a result similar to the BEx query, you must move the `Sales Organization` dimension to the rows axis.

```
commandProcessor.moveDimensionToRows(salesOrgDimension);
```

Step 6: Specify Selected Members

To further specialize the query, use the OLAP Command Processor to specify the members you would like to select for some of the dimensions.

In this case, you must add the member `HiTech` to the member set of the `Division` dimension. Because this dimension is still on the slicer axis, this dimension results in a single filter value in the `WHERE` clause of the resulting MDX:

```
commandProcessor.addMember(dimensionHiTechMember);
```

Step 7: Execute the Query and Retrieve the Result Set

Now you can execute the query and retrieve its result set, by calling the `execute()` method on a query object. This method triggers the selection of data from the connected server:

```
IBIDataSet dataset = query.execute();
```

Instances of `IBIDataSet` represent data sets, which are multidimensional result sets.



Note:

For more information on the SDK's `ResultSet` API, see the [Retrieving Result Sets](#) chapter of this Guide.

Step 8: Render the Result Set

With your query created, you can visualize its result by rendering it into the output stream of a servlet. To format the HTML, enlist the help of a stylesheet contained in the examples helpers class `Helpers.java`, and for illustration, also display the MDX statement that was executed on the server:

Hello MDX: First Example for a Multidimensional Data Source

```
out.println(Helpers.getDocTypeDefinition());
out.println("<html>");
out.println("<head><title>Tutorial_1</title>");
out.println(Helpers.getStyleSheetDefinition());
out.println("</head><body>");

out.println(
    "<p><b>MDX Statement that was executed:</b><br> "
    + "<span class=\"code\">"
    + (String) query.getNativeRepresentation()
    + "</span>"
    + "</p>");
```

Now you use the OLAP Table Model, `BIDataSetTableModel`, in conjunction with the stylesheet to render to a table. This provides a projection of the data set into a tabular view where cells can be accessed by row and column coordinates:

```
try {
    BIDataSetTableModel table =
        new BIDataSetTableModel(dataset, false);
    out.println("<p>Result set:</p>");
    out.println("<table width=700 border=1 cellpadding=0 cellspacing=0>");

    int row = table.getRowCount();
    int col = table.getColumnCount();

    for (int i = 0; i < row; i++) {
        out.println("<tr>");

        for (int j = 0; j < col; j++) {
            BITableItem item =
                (BITableItem) table.getValueAt(i, j);
            out.println("<td class=\"headCenter\">");
            out.println(Helpers.escape(item.toString()));
            out.println("</td>");
        }

        out.println("</tr>");
    }

    out.println("</table>");
} catch (BISQLException e) {
    e.printStackTrace();
    out.println("<p>Error: " + e.getMessage() + "</p>");
}

out.println("</body>");
out.println("</html>");
}
```



Note:

We've kept the HTML fairly simple in this tutorial, but as you will see in the other examples, you can use the stylesheet to distinguish between header rows and columns and to stripe rows of the data.

And finally, catch errors and finish the servlet:

```
catch (Exception e) {
    e.printStackTrace();
    if (e instanceof IBaseException)
        out.println("Error: " +
            ((IBaseException)e).getNestedLocalizedMessage());
    else
        out.println("Error: " + e.getMessage());
}
}

public void destroy() {
}

public static void main(String[] args) {
    if (args.length == 1) {
        MinimalServletContainer.executeServlet(new Tutorial_1(), args[0]);
    } else {
        MinimalServletContainer.executeServlet(new Tutorial_1(), System.out);
    }
}
}
```



Note:

See more on the SDK's exception framework in the Exceptions chapter of this Guide.

The Output

The output of this tutorial is the following HTML table:

Hello SQL: First Example for a Relational Data Source

MDX Statement that was executed:

```
SELECT NON EMPTY [Measures].MEMBERS ON AXIS(0), NON EMPTY [0D_SALE_ORG].MEMBERS ON
AXIS(1) FROM [0D_SD_C03/0D_SD_C03_Q009] WHERE ([0D_DIV].[7]) CELL PROPERTIES
CELL_ORDINAL, VALUE, FORMATTED_VALUE
```

Result set:

	Incoming orders value	Billing value	Returns value	Credit memo value
All Sales organization	194,174,797.58 *	186,846,004.29 *	11,899,068.03 *	8,610,832.16 *
Frankfurt	7,551,870.83 EUR	7,278,661.82 EUR	205,535.69 EUR	169,745.46 EUR
Berlin	11,445,868.36 EUR	11,215,095.32 EUR	291,228.97 EUR	218,216.68 EUR
Munich	22,297,907.43 EUR	20,821,494.95 EUR	1,227,705.02 EUR	1,213,388.94 EUR
Hamburg	6,388,127.30 EUR	6,205,290.09 EUR	492,882.69 EUR	490,019.07 EUR
London	£ 14,984,849.00	£ 14,569,213.00	£ 679,660.00	£ 354,560.00
Birmingham	£ 9,504,655.00	£ 9,272,423.00	£ 423,060.00	£ 207,440.00
Paris	29,869,563.66 EUR	28,069,450.11 EUR	2,513,570.66 EUR	1,536,671.01 EUR
New York	\$ 24,722,543.00	\$ 24,109,951.00	\$ 1,574,226.00	\$ 1,547,560.00
San Francisco	\$ 8,550,079.00	\$ 7,917,756.00	\$ 718,226.00	\$ 718,226.00
Toronto	31,519,421.00 CAD	30,570,831.00 CAD	2,497,600.00 CAD	1,364,400.00 CAD
Vancouver	25,598,113.00 CAD	25,166,071.00 CAD	1,218,110.00 CAD	704,710.00 CAD
Sydney	1,741,800.00 AUD	1,649,767.00 AUD	57,263.00 AUD	85,895.00 AUD

Figure 3 — Tutorial 1 Result Set

Remember, you can view the full source code for this tutorial and the `Helpers` class in `Tutorial_1.java` in the `docs/examples` folder after unzipping the BI Java SDK. There, you can also view the HTML output of the above result.

Hello SQL: First Example for a Relational Data Source

Our SQL version of “Hello World” demonstrates the complete code sequence required to display a result on the screen. In this tutorial, you build a servlet that connects to a relational data source, accesses its metadata, builds a query, executes it, and accesses and renders the result set. You will actually be defining and rendering the result of a SQL query, but with our Relational Query API, the complexity of the underlying SQL statement is hidden.

This tutorial is a little different from the OLAP tutorial in that it creates a query against the data in your own JDBC database, rather than using specific data in a specific data source such as a BI cube. We will show you how to retrieve the data in all the columns of the first table in your JDBC database, so results on different databases will vary.

Step 1: Import Packages

First, configure the imports required for this SDK servlet. You may import by package, but we recommend you import by specific interfaces. For this tutorial, you will need the basic data access, connectivity, query generation, and supporting interfaces. The full list of imports is described below:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.ResultSet;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;

import javax.resource.spi.ManagedConnectionFactory;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.omg.cwm.resource.relational.Column;
import org.omg.cwm.resource.relational.Table;

import com.sap.exception.IBaseException;
import com.sap.ip.bi.sdk.dac.connector.IBIConnection;
import com.sap.ip.bi.sdk.dac.connector.IBIRelational;
import com.sap.ip.bi.sdk.dac.connector.jdbc.JdbcConnectionConstants;
import com.sap.ip.bi.sdk.dac.connector.jdbc.JdbcManagedConnectionFactory;
import com.sap.ip.bi.sdk.dac.relational.query.IBICommandProcessor;
import com.sap.ip.bi.sdk.dac.relational.query.IBIQuery;
import com.sap.ip.bi.sdk.samples.servlet.MinimalServletContainer;
import com.sap.ip.bi.sdk.util.impl.BIResourceProperties;
import com.sapportals.connector.connection.IConnectionFactory;
import com.sapportals.connector.connection.IConnectionSpec;
```

Step 2: Connect to a JDBC Database

The next step is to connect to a JDBC database. A connection to a data source is represented by an instance of the `IBIConnection` interface.

First create a basic servlet and prepare to read the connection information from the properties helper file, with help from `Helpers.java`:

Hello SQL: First Example for a Relational Data Source

```
public class Tutorial_2 extends HttpServlet {
    private final static String CONTENT_TYPE = "text/html";

    private static Properties connProp =
        new BIResourceProperties(Helpers.class, ".jdbc");

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType(CONTENT_TYPE);

        PrintWriter out = response.getWriter();
```

To establish a connection, you must provide connection information in your `Helpers.jdbc.properties` file that encapsulates all the relevant connection properties for your JDBC database.



Note: About JDBC connection URLs

The URL used to connect to a JDBC database is vendor-specific. JDBC URLs have the following components:

`<protocol>:<subprotocol>:<subname>`

where `<protocol>` is always "jdbc." Both `<subprotocol>` and `<subname>`, however, are vendor-specific, depending on the actual JDBC driver used.

Get the connection properties with which to create the connection from the properties file with the following lines:

Hello SQL: First Example for a Relational Data Source

```
try {
    ManagedConnectionFactory mcf;
    IConnectionFactory cf;
    IConnectionSpec cs;

    mcf = new JdbcManagedConnectionFactory();
    cf = (IConnectionFactory) mcf.createConnectionFactory();
    cs = cf.getConnectionSpec();

    cs.setPropertyValue(
        JdbcConnectionConstants.USERNAME.toString(),
        connProp.getProperty(JdbcConnectionConstants.USERNAME.toString()));

    cs.setPropertyValue(
        JdbcConnectionConstants.PASSWORD.toString(),
        connProp.getProperty(JdbcConnectionConstants.PASSWORD.toString()));

    cs.setPropertyValue(
        JdbcConnectionConstants.URL.toString(),
        connProp.getProperty(JdbcConnectionConstants.URL.toString()));

    cs.setPropertyValue(
        JdbcConnectionConstants.DRIVERNAME.toString(),
        connProp.getProperty(JdbcConnectionConstants.DRIVERNAME.toString()));
}
```

**Note:**

Be sure to prepare the `Helpers.jdbc.properties` file with your own connection parameters.

Actually create the connection now by instantiating `IBIConnection`. In establishing the connection to a relational database, the `IBIRelational` interface provides an entry point to access metadata and execute queries:

```
IBIConnection connection = (IBIConnection) cf.getConnectionEx(cs);
IBIRelational rel = connection.getRelational();
```

**Note:**

For more information on the SDK's connection architecture, see the [Connecting to Data Sources](#) chapter of this Guide.

Step 3: Retrieve the Metadata

Standard client applications need access to metadata to construct queries. Now that you're connected to your data source, you will retrieve its metadata. In this example, you retrieve metadata by using the `getTables` method provided by the `IBIRelational` interface.

First, you get an interface to a table in your database by getting a list of all tables, then retrieving the first table in the list. You then retrieve the list of columns of that table, evoking exceptions if no tables or columns are found:

```
List tables = rel.getTable();
if (tables == null || tables.size() == 0) {
    throw new ServletException("no tables found");
}

Table table = (Table) tables.get(0);

List columns = table.getFeature();
if (columns == null || columns.size() == 0) {
    throw new ServletException("no columns found");
}
```

**Note:**

For more information on the SDK's metadata APIs, see the [Accessing Metadata](#) chapter of this Guide.

Step 4: Create a Query

You're now ready to create a query. Query construction is done by `rel.createQuery()`, and `IBIQuery` is the outcome.

Querying is greatly assisted by the use of the Relational Command Processor. The Relational Command Processor is part of the Relational Query API and makes it easier to use the underlying query model by hiding the complexity of this model. With it, you can create and manipulate complex queries with simple commands. You can think of the individual methods of the command processors in terms of macros that consist of several method calls manipulating the structures of queries.

Next, you therefore create a query and an instance of the Relational Command Processor associated with this query:

```
IBIQuery query = rel.createQuery();
IBICommandProcessor commandProcessor = query.getCommandProcessor();
```

Hello SQL: First Example for a Relational Data Source

Although a query can be executed immediately after its creation, executing it would not be very useful because the query does not contain specific information. No tables and columns are specified, and such a query would return data in the list of all the tables and all the columns in your database. Refine your query, therefore, by specifying table and columns.

**Note:**

For more information on the SDK's query APIs, see the [Creating Queries](#) chapter of this Guide.

Step 5: Specify Table and Columns

Since relational queries support multiple tables and even multiple instances of the same table in the same query, each instance of a table in the query has to be uniquely identifiable. This is achieved by using references (`String`) identifying the particular instance of the table in the query. These references can either be provided when adding the table or generated by the command processor when adding the table.

In this tutorial, you use the Relational Command Processor to add the table to the query:

```
String tref = commandProcessor.addTable(table);
```

Continue using the command processor to add all of the columns of the table to the query:

```
for (Iterator c = columns.iterator(); c.hasNext();) {  
    Column column = (Column) c.next();  
    commandProcessor.addColumn(column, tref);  
}
```

Step 6: Execute the Query and Retrieve the Result Set

Now you can execute the query and retrieve its result set, by calling the `execute()` method on a query object. This method triggers the selection of data from the connected relational database:

```
ResultSet result = query.execute();
```

The `ResultSet` object represents the relational result set.

**Note:**

For more information on the SDK's `ResultSet` API, see the [Retrieving Result Sets](#) chapter of this Guide.

Step 7: Render the Result Set

With your query created, you can visualize its result by rendering it into the output stream of a servlet. To format the HTML, enlist the help of a stylesheet contained in the examples helpers class `Helpers.java`, and for illustration, also display the SQL statement that was executed on the server:

```
out.println(Helpers.getDocTypeDefinition());
out.println("<html>");
out.println("<head><title>Tutorial_2</title>");
out.println(Helpers.getStyleSheetDefinition());
out.println("</head><body>");

out.println(
    "<p><b>SQL Statement that was executed:</b><br> <code>"
    + (String)rel.getNativeRepresentation(query)
    + "</code><p>");

Helpers.renderResultSet(out, result);

out.println("</body>");
out.println("</html>");
}
```

And finally, catch errors and finish the servlet:

```
catch (Exception e) {
    e.printStackTrace();
    if (e instanceof IBaseException)
        out.println("Error: " +
            ((IBaseException)e).getNestedLocalizedMessage());
    else
        out.println("Error: " + e.getMessage());
}

public void destroy() {
}

public static void main(String[] args) {
    if (args.length == 1) {
        MinimalServletContainer.executeServlet(new Tutorial_2(), args[0]);
    } else {
        MinimalServletContainer.executeServlet(new Tutorial_2(), System.out);
    }
}
}
```



Note:

See more on the SDK's exception framework in the Exceptions chapter of this Guide.

The Output

This tutorial fetches all the columns for the first table retrieved in your own JDBC database, so your results will vary. The output against one example JDBC database is the following HTML table:

SQL Statement that was executed:

```
SELECT "t1"."MANDT", "t1"."ID", "t1"."NAME", "t1"."POSTCODE", "t1"."CITY",  
"t1"."CUSTTYPE", "t1"."DISCOUNT", "t1"."TELEPHONE" FROM "TEST_CUSTOMERS" "t1"
```

Result set:

MANDT	ID	NAME	POSTCODE	CITY	CUSTTYPE	DISCOUNT	TELEPHONE
000	00000001	Edwards		Small Town		000	654-321-1234
000	00000002	Young		Village		000	333-444-2222
000	00000017	Smith		Big City		000	717-161-5151
000	87654321	Edison		Market Place		000	111-111-1111

Figure 4 — Tutorial 2 Sample Result Set

Remember, you can view the full source code for this tutorial and the `Helpers` class in `Tutorial_2.java` in the `docs/examples` folder after unzipping the BI Java SDK. There, you can also view the HTML output of the above result.

Chapter 2: Connecting to Data Sources

Overview

The first step in building applications with the BI Java SDK is to connect to a data source. The BI Java SDK relies upon a driver-based connector architecture based on the J2EE Connector Architecture (JCA) in order to achieve the openness necessary to integrate many different data sources into a variety of Java applications and BI tools.

This chapter explains the architecture and basic concepts involved with connecting to data sources with the SDK, in the following sections:

- Connection Architecture
 - Client Interface
 - Portal Connection Framework
 - Service Provider Interface
 - Managed Environment
 - Non-Managed Environment
 - Connection Specification and Portal Service
- BI Java Connectors
 - BI XMLA Connector
 - BI JDBC Connector
 - BI ODBO Connector
 - BI SAP Query Connector



Note:

To find out more about the JCA and SAP NetWeaver Portal Architecture, refer to references listed in Appendix C: Additional Resources.

API Documentation:

Refer to the Javadocs for the Connection Interfaces in the following package of the SDK:

com.sap.ip.bi.sdk.dac.connector

Connection Architecture

Sun's J2EE Connector Architecture (JCA) defines a standard architecture for connecting the Java 2 Platform to heterogeneous Enterprise Information Systems (EISs) such as mainframe transaction processing and database systems. The JCA enables an EIS vendor to provide a standard resource adapter, in our case a BI Java Connector, which is defined as a system-level software driver used by a Java application to connect to an EIS. The connector plugs into an application server and provides connectivity between the EIS, the application server, and the application.

The BI Java Connectors implement JCA's Service Provider Interface (SPI) to realize the system contract. The connectors use the connection management components of both the JCA's Common Client Interface (CCI) and the SPI. In a layered approach, the BI Java Connectors implement the connection management interfaces in an abstract implementation layer, and each individual connector implements a concrete layer.

The figure below details this layered connection management approach:

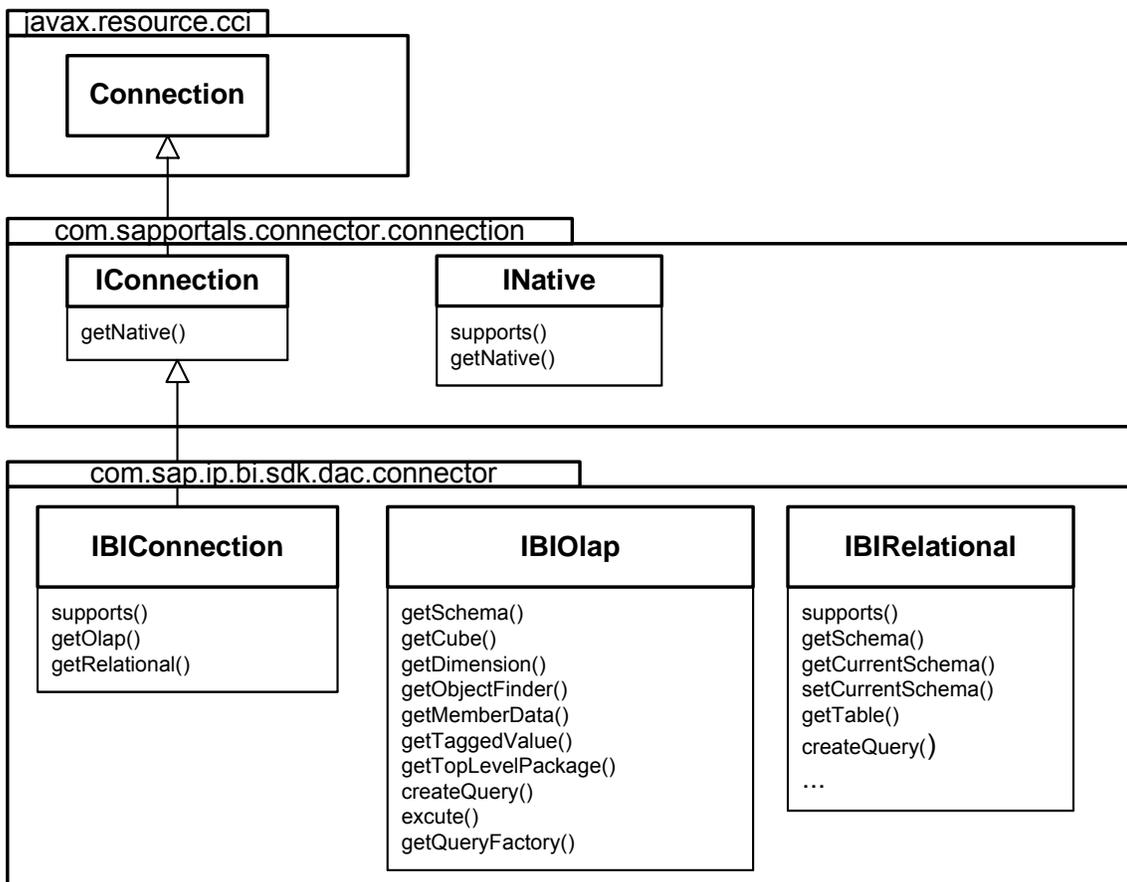


Figure 5 — BI Java SDK Connection Management

Connection Architecture

In brief, as depicted, JCA's CCI provides the `Connection` interface, which the SAP NetWeaver Portal has extended with the `IConnection` interface of its Portal Connection Framework API. Taking this one step further, for BI-specific connections, we've extended `IConnection` with our `IBIConnection` interface. By extending the Portal Connection Framework interfaces and thus integrating with the Portal Runtime, Java components can leverage services provided by the Portal such as `SystemLandscape` and `ConnectorGateway`.

Client Interface

The Common Client Interface (CCI) is a key component of the JCA specification. This is the interface that a resource adapter – in our case, a BI Java Connector – provides to a client application to enable it to interact with its underlying EIS. This interface defines a standard client API for application components, which enables application components and enterprise architecture frameworks to drive interactions across heterogeneous EISs using a common client API.

Each BI Java Connector used by the BI Java SDK implements the CCI interfaces described below:

- `IConnectionFactory` – the extension of `javax.resource.cci.ConnectionFactory`.
- `IConnectionSpec` – the extension interface of `javax.resource.cci.ConnectionSpec`.
- `IConnection` – the extension of `javax.resource.cci.Connection` interface. In the BI Java SDK, we create an `IBIConnection` interface that extends the `IConnection` interface. `IBIRelational` stands for tabular databases, and `IBIOlap` stands for multidimensional data sources.
- `IConnectionMetaData` – the extension of interface `javax.resource.cci.ConnectionMetaData`. A component calls the method `IConnection.getMetaData` to get a `ConnectionMetaData` instance. A CCI implementation is required to provide an implementation class for the `ConnectionMetaData` interface.



Note:

Refer to the JCA specification, at <http://www.jcp.org/en/jsr/detail?id=16>, for documentation on these interfaces.

Consumer code deals only with the external CCI interfaces; these interfaces provide a wrap encapsulating all other JCA interfaces.

In short, here's how to connect to data sources using the SDK:

1. In your consumer code, look up the CCI connection factory interface of the connector.
2. Create a connection specification (connection string) object and set the connection parameters, including user context. This may be done using a predefined connection string related to this connector. See the `howto.html` file inside each BI Java Connector's resource adapter archive for a list of specific connection parameters.
3. Call the `getConnection` method that receives the `connectionSpec` object as a parameter.
4. A CCI connection interface is returned as a result of the call.

Portal Connection Framework

The SAP NetWeaver Portal has extended the CCI with its Portal Connection Framework interface, which consists of three components: Connection, Execution, and Metadata. The BI Java SDK uses this Portal Connection Framework, but only relies upon its Connection component. The Execution component defines functions, objects, and structures not currently used by the SDK, and for the Metadata functionality, the SDK has leveraged the CWM and JMI specifications (see Accessing Metadata, below).

The Connection component of the Portal Connection Framework provides the connection management interfaces (again, extending CCI's connection component), which include:

- `IConnection` – the extension of CCI's `Connection`
- `IConnectionFactory` – the extension of CCI's `ConnectionFactory`
- `IConnectionMetaData` – the extension of CCI's `ConnectionMetaData`
- `IConnectionSpec` – the extension of CCI's `ConnectionSpec`
- `INative` – the native handle to an EIS

Service Provider Interface

The BI Java SDK's Service Provider Interface (SPI) handles connection management contracts between an application server and a resource adapter.

Below are the service provider interfaces which need to be implemented by a resource adapter:

- `ConnectionManager` - `javax.resource.spi.ConnectionManager`
- `ManagedConnectionFactory` - `javax.resource.spi.ManagedConnectionFactory`
- `ManagedConnection` - `javax.resource.spi.ManagedConnection`
- `ManagedConnectionMetaData` - `javax.resource.spi.ManagedConnectionMetaData`

(Refer to the JCA specification, at <http://www.jcp.org/en/jsr/detail?id=16>, for documentation on these interfaces.)

Managed Environment

SAP's BI resource adapters, the BI Java Connectors, may be used without an application server (in a non-managed environment), or may be deployed onto SAP's J2EE-based application server, the Web Application Server, version 6.40 (in a managed environment). The Web Application Server (WAS) is a J2EE application server that complies with JCA.

During the deployment process, the Web Application Server binds the Portal Connection Framework connection factory (IConnectionFactory) of the BI Java Connector to a JNDI lookup string. When the application server starts up, it instantiates a managed connection factory, passing an instance of its own connection manager to the BI Java Connector.

This process is schematically diagrammed below:

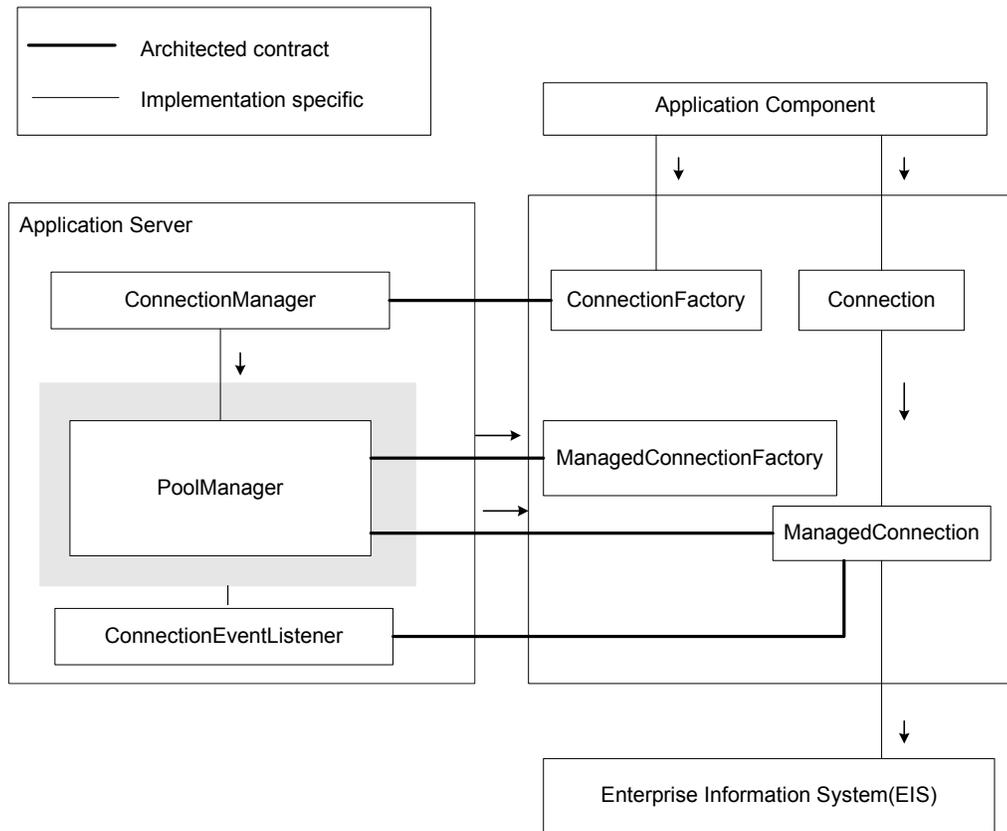


Figure 6 — Architecture Diagram: Managed Application Scenario

Non-managed Environment

As stated above, the BI Java Connectors may also be used without an application server (in a non-managed environment). In this environment, your application client (the first tier) directly uses a BI Java Connector to access the EIS, which defines the second tier for a two-tier application. The BI Java SDK contains the JAR files you need to develop applications using any of the BI Java Connectors and to use them in an unmanaged environment.

Connection Architecture

In this scenario, the BI Java Connector uses its default connection manager. Your consumer code creates an instance of the Portal Connection Framework `IConnectionFactory` without passing a connection manager as parameter. The `IConnectionFactory` creates a managed connection factory and the Connector's default connection manager.

Since the BI Java Connector is deployed in this scenario to an environment that does not contain the application server end of JCA, the managed connection factory, implemented by each individual BI Java Connector, is directly exposed to the consumer.

You can instantiate an instance of the managed connection factory and work directly with the physical connection layer to manage connections.

Connection Specification and Portal Service

You need to specify certain parameters in order to configure your connection. The SDK implements the empty `ConnectionSpec` interface provided by the JCA for these purposes. In addition, the BI Java Connectors use the `ConnectionSpec` to define additional properties specific to the underlying EIS.

In order to conform to the Portal Connection Framework, we implement `IConnectionSpec`, which is an extension of `ConnectionSpec`.

In order to use Portal services such as the `ConnectorGateway`, we also need to implement several Portal-specific interfaces, including:

- `IConnectionSpecMetaData`
- `IConnectionProperty`
- `IConnectionPropertyGroup`
- additional related implementations such as `IString`

The following diagram illustrates `ConnectionSpec` and its related classes, using the BI ODBO Connector scenario as an example:

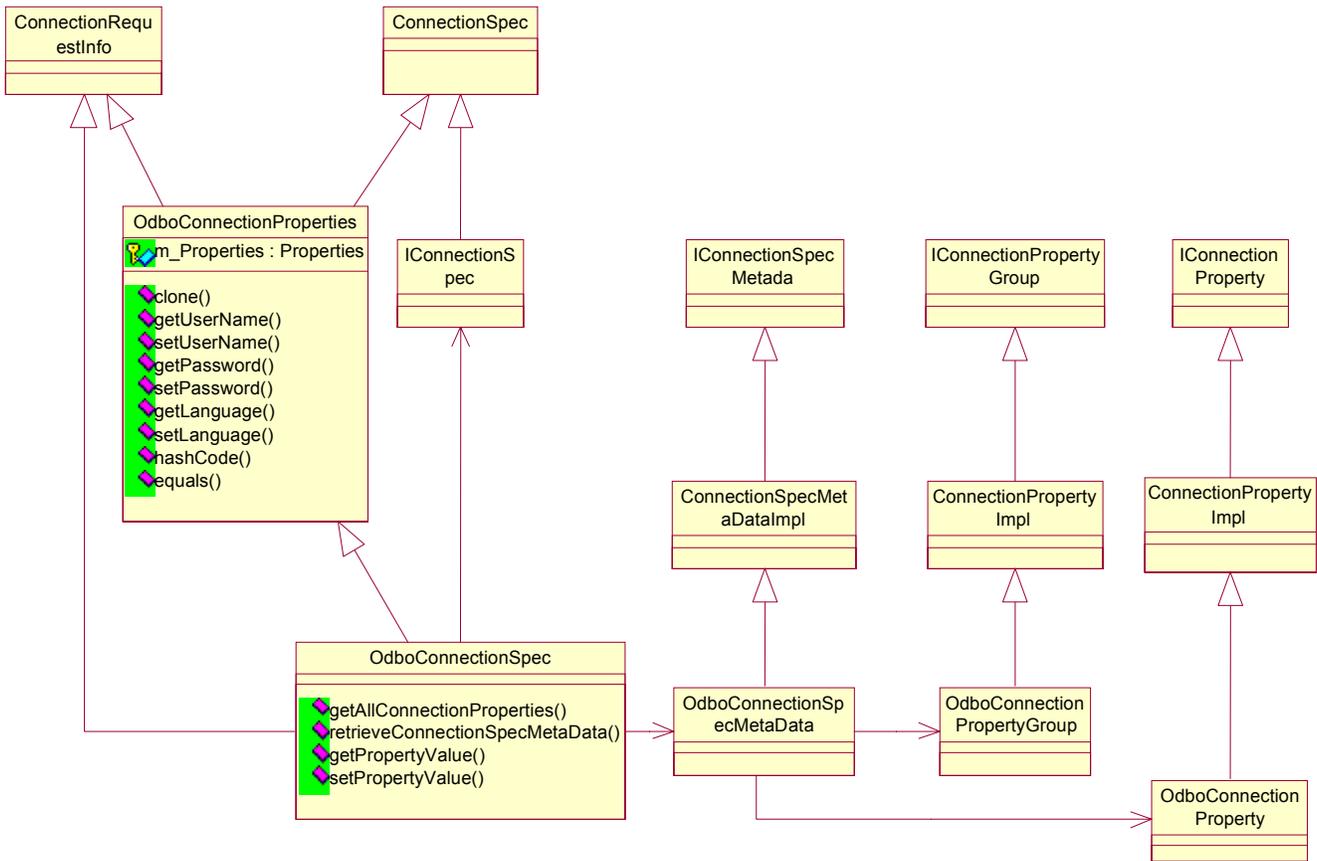


Figure 7 — ConnectionSpec and Related Classes

BI Java Connectors

In the JCA architecture, a resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the application.

The BI Java Connectors are a group of four JCA-compliant resource adapters created for Java applications to connect to relational or OLAP data sources in an EIS. The BI Java Connectors may be deployed onto SAP NetWeaver '04 - Web Application Server version 6.40. The connectors implement the specific client interface defined by the SDK's Connector Architecture.

The BI Java SDK contains the JAR files you need to develop applications using any of the BI Java Connectors and to use them in an unmanaged scenario, but to use your application with a data source in the managed environment of the J2EE server, you need to deploy the appropriate BI Java Connector. The connectors themselves are distributed separately, deployed with NetWeaver's optional BI UDI component.

BI Java Connectors

The BI Java Connectors are packaged in resource adapter archives, or RAR files. Each RAR file includes class libraries and dependencies, a deployment descriptor, and connector properties documentation in the form of a `howto.html` file. General system guidelines are listed below in the Connector Overview section.

 **Note:**

The connectors' `howto.html` files are also included in the SDK distribution package for your reference. See [index.html](#) in the docs folder of the package, then follow the Connectors link.

Four BI Java Connectors are available, listed below with the name of the resource adapter archive in which they are deployed:

- BI JDBC Connector : `bi_sdk_jdbc.rar`
- BI ODBO Connector : `bi_sdk_odbo.rar`
- BI SAP Query Connector : `bi_sdk_sapq.rar`
- BI XMLA Connector : `bi_sdk_xmla.rar`

The flow between the data sources, the BI Java Connectors, the BI Java SDK, and its potential clients is schematically depicted below:

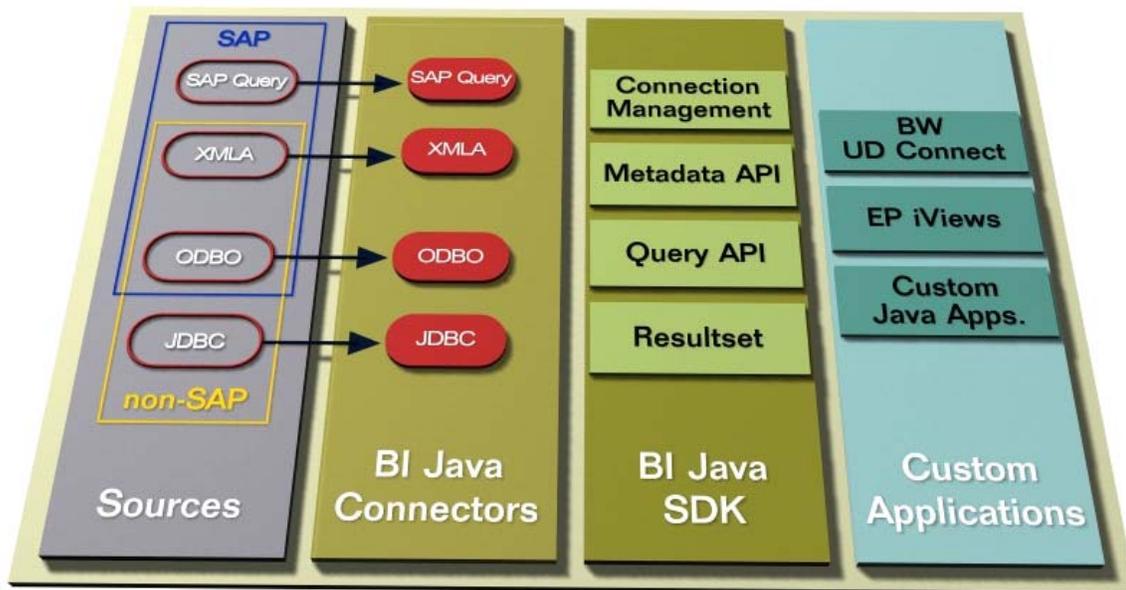


Figure 8 — BI Java SDK Connectivity Flow

BI Java Connectors

The following sections briefly introduce each connector, but refer to `howto.html` file inside of each archive for additional specific configuration and deployment information.



Note:

The BI Java Connectors are distributed separately from the BI Java SDK, deployed with NetWeaver's optional BI UDI component.

BI JDBC Connector

Sun's JDBC (Java Database Connectivity) is the standard Java API for relational database management systems (RDBMS). The BI JDBC Connector allows you to connect applications built with the BI Java SDK to over 170 JDBC drivers, supporting data sources such as Teradata, Oracle, Microsoft SQL Server, Microsoft Access, DB2, Microsoft Excel, and text files such as CSV.

The connector adds the following functionality to existing JDBC drivers:

- Uniform connection management that integrates with user management in the SAP NetWeaver Portal
- Uniform metadata service, by implementing Java Metadata Interface (JMI) capabilities based on the Common Warehouse Metamodel (CWM)
- SQL generator

The BI JDBC Connector implements the BI Java SDK's `IBIRelational` interface.

BI ODBO Connector

Microsoft's ODBO (OLE DB for OLAP) is the established industry-standard OLAP API for the Windows platform. The BI ODBO Connector allows you to connect applications built with the BI Java SDK to ODBO-compliant OLAP data sources such as Microsoft Analysis Services, SAS, Microsoft PivotTable Services, and SAP's BI.

The BI ODBO Connector uses Microsoft's ADO (ActiveX Data Objects) and ADO MD (ActiveX Data Objects Multidimensional) to support connectivity to OLAP data sources. ADO provides access to the schema object; ADO MD adds easy access to multidimensional data by extending ADO with objects specific to multidimensional data, such as the cubes and cellsets. With ADO and ADO MD, you can browse multidimensional schema, query a cube, and retrieve the results, thus providing convenient access to OLAP data from languages such as Microsoft Visual Basic, Microsoft Visual C++, and Microsoft Visual J++. Like ADO, ADO MD uses an underlying OLE DB provider to gain access to data.

The BI ODBO Connector implements the BI Java SDK's `IBIOlap` interface.

Examples

BI SAP Query Connector

SAP Query is a component of SAP's Web Application Server that allows you to create custom reports without any ABAP programming knowledge. The BI SAP Query Connector uses SAP Query to allow applications created with the BI Java SDK to access data from these SAP operational applications.

The BI SAP Query Connector implements the BI Java SDK's `IBIRelational` interface.

BI XMLA Connector

Microsoft's XMLA (XML for Analysis) facilitates Web services-based, platform-independent access to OLAP providers. The BI XMLA Connector enables the exchange of analytical data between a client application and a data provider working over the Web, using a SOAP-based XML communication API. The XMLA Connector sends commands to an XMLA-compliant OLAP data source in order to retrieve the schema rowsets and obtain a result set.

The BI XMLA Connector allows you to connect applications built with the BI Java SDK to data sources such as Microsoft Analysis Services, Hyperion, MicroStrategy, MIS, and SAP BI 3.x.

The BI XMLA Connector implements the BI Java SDK's `IBIOlap` interface.

Examples

Although all of the SDK's examples connect to data sources, the tutorial examples focus in particular on connecting, only minimally referring to the `Helpers` class to retrieve connection properties:



Tutorial_1.java – OLAP Tutorial:

The OLAP Tutorial specifically steps you through setting connection properties and connecting to an OLAP data source – in this case, via the BI XMLA Connector. This tutorial reads your connection properties from a properties file and connects to your data source with the assistance of helper methods in `Helpers.java`.

See Hello MDX: First Example for a Multidimensional Data Source in the Getting Started chapter for step-by-step instructions on how to use this tutorial.



Tutorial_2.java – Relational Tutorial:

Examples

The Relational Tutorial specifically steps you through setting connection properties and connecting to a relational database – in this case, using the BI JDBC Connector. This tutorial reads your connection properties from a properties file and connects to your database with the assistance of helper methods in `Helpers.java`.

See Hello SQL: First Example for a Relational Data Source in the Getting Started chapter for step-by-step instructions on how to use this tutorial.



Helpers.java:

The `Helpers` class provides static helper methods that facilitate connecting to data sources, reading your own connection properties from `Helpers.*.properties` files.



Note:

See Appendix B: Examples for the full index of examples and instructions on getting your system up and running with them.

Chapter 3: Accessing Metadata

Overview

Once the connector has established a connection to the data source, the next step is to understand the data inside of it in order to browse the metadata with which to build queries. To do this, the SDK's connectors rely upon metadata APIs, which provide interfaces that expose the metadata of a given data source. These APIs are rendered via the Java Metadata Interface (JMI) from standard metadata models provided by the Common Warehouse Metamodel (CWM), and each in turn provide the basis upon which the query APIs can formulate queries.

This chapter describes the CWM, generation of interfaces from models via JMI, and the resultant metadata APIs of the SDK, including the models that form their basis, in the following sections:

- Common Warehouse Metamodel
- Generating Interfaces
- Metadata APIs
 - OLAP Metadata Model
 - Relational Metadata Model

API Documentation:

Refer to the Javadocs for the OLAP Metadata API in the following package of the CWM:

org.omg.cwm.analysis.olap

Refer to the Javadocs for the Relational Metadata API in the following package of the SDK:

org.omg.cwm.resource.relational

Common Warehouse Metamodel

A metadata model defines an abstract language for expressing metadata, or data about data. In order to support connectivity wide ranges of data sources, the SDK needs to rely upon metadata models that have sufficient expressiveness to cover a broad range of different OLAP and relational implementations. To do this, the SDK uses the Common Warehouse Metamodel (CWM), which provides a basis for common metadata exchange.

CWM is an Object Management Group (OMG) open standard that describes the representation and exchange of shared, global metadata in the data warehousing and analysis, business intelligence, knowledge management, and

Generating Interfaces

portal technologies domains. The SDK has leveraged CWM to create its metadata models. CWM provides a framework for representing metadata about data sources, data targets, transformations and analysis, and the processes and operations that create and manage warehouse data and provide lineage information about its use. The CWM Metamodel consists of a number of sub-metamodels, which represent common warehouse metadata in data warehousing, business intelligence, knowledge management, and portal technologies.

CWM metamodels are capable of modeling a wide spectrum of data providers. The metamodels are not only generic and extensible in their overall content and structure, but are also separated from implementation considerations. This is of particular importance for the SDK, in which we support a large variety of providers mapped into common metamodel.

Our use of CWM is fully in keeping with the standards-based approach we outline in the Introduction, in Open Standards in the SDK. Using CWM's industry-standard metamodels means we use their respective terminology, which is agreed upon by a wide spectrum of vendors active in the data warehousing domain. This in turn makes it easy even for a non-SAP BI specialist to understand and learn the organization of the BI Java SDK's APIs, because they build upon this common industry domain knowledge.

The CWM metamodels also provide excellent documentation with precise definitions of all objects, attributes, and methods, and package-level documentation that introduces the major concepts of a specific sub-model. We provide that documentation in our distribution package, and cite it in this chapter.

Generating Interfaces

A large amount of the APIs provided by the BI Java SDK, including its metadata APIs, are generated by a template-based translation or rendering of CWM models into Java interfaces. In order to generate Java interfaces out of these models, the SDK uses Java Metadata Interface (JMI) mapping. JMI is an extensible metadata service for the Java platform, used for introspection, discovery, and dynamic access to metadata sources, thus creating a Java programming model for accessing metadata. It supports the design and use of generic tools for working with metadata, for example the interaction with a metadata repository such as SAP's Metamodel Repository.

In this way, CWM metamodels are easily extensible in that they are separated from implementation considerations, since the implementation is rendered separately through the use of JMI.

A JMI service is any system that provides a JMI-compliant API to access its public metadata. The BI Java Connectors expose metadata of the underlying EIS via JMI services. Any type of metadata, including objects that form the basis of OLAP and relational queries, can be represented via JMI-compliant interfaces and implementations that are generated by the JMI mapping service of the Metamodel Repository. JMI thus provides the benefits of interoperability and compliance with the industry standards, as well as allowing for future extensibility.

Through JMI, the metadata models are translated into Java APIs that allow your visualization components to access and navigate instances of the metamodels. The process is diagrammed below:

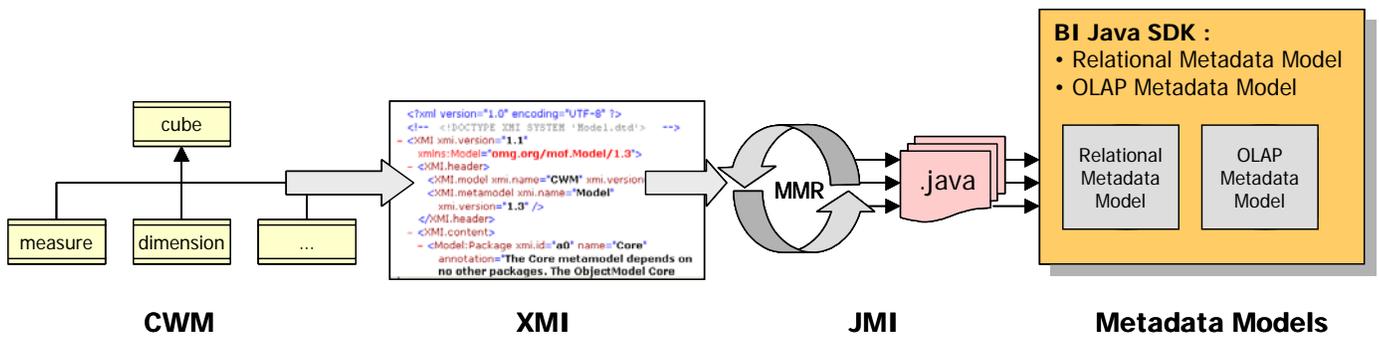


Figure 9 — The JMI Process

As diagrammed, the major contributors to this process are:

1. **CWM:** The SDK uses Unified Modeling Language (UML) tools to extend CWM models with our Business Intelligence needs.

The SDK has built upon the basis of CWM to create its metadata models. CWM provides standard base UML metamodels in UML format. UML represents the model, and the Meta Object Facility (MOF) provides the modeling language, including syntax and semantics.

2. **XMI:** The UML file is saved in XMI format and ready for interchange.

XML Metadata Interchange (XMI) is the interchange mechanism for the modeled metadata, helping exchange it from its origin in UML into its destination as Java source code.

3. **JMI:** We use the MMR to create a discrete Java interface for each modeled metadata object.

SAP's Metamodel Repository (MMR) is a tool that uses JMI to create Java interfaces out of the XMI file.

4. **Metadata Models:** The SDK delivers these in its Java APIs.

The metadata models can be thought of as templates for creating objects, which the connectors then use to create instances of the metadata objects to expose to your application components:

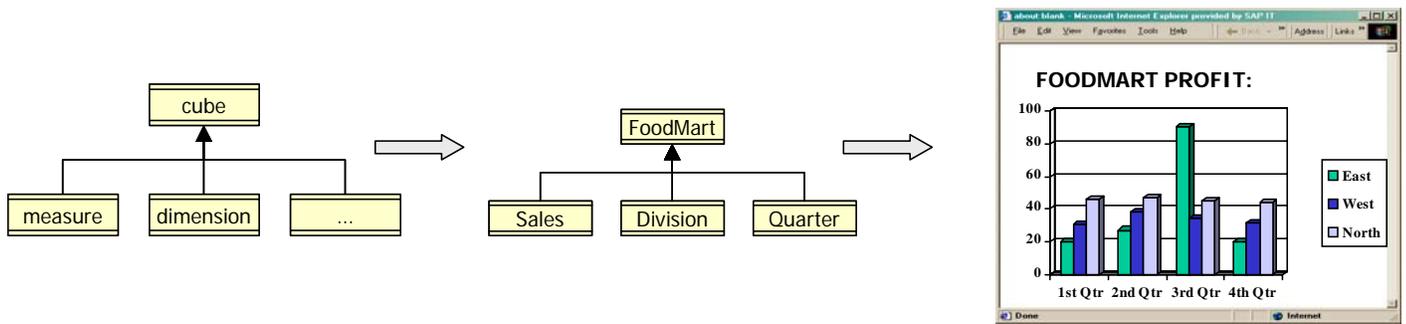


Figure 10 — UML Metadata "Templates"

The UML diagram on the left is the template for creating the objects (center) which are then exposed to your application components (right).

 **Caution:**

Metadata instances referring to the same metadata object retrieved from the same data source are guaranteed to be identical (a compare with "==" would deliver "true") only when they have been retrieved via the same connection instance.

This means when two connections to the same data source are open at once, it is possible that metadata instances with the same MOF-ID referring to the same metadata object of the data source are not identical in the Java sense. In other words, comparing two such metadata objects with "equals()" might result in "true", whereas comparing them with "==" might not.

 **Note:**

The JMI rendering process also includes the model description as Javadocs, and as part of our documentation set, we have included the Javadocs for the CWM packages used by the BI Java SDK. Specific references are cited in this chapter.

Metadata APIs

The SDK's metadata APIs are rendered via JMI from metadata models. The SDK provides two metadata models, each based on the Common Warehouse Metamodel (CWM): the OLAP Metadata Model, and the Relational Metadata Model. These metadata models represent all the metadata objects necessary to create queries upon a wide variety of data sources – both relational and OLAP.

OLAP Metadata Model

The OLAP Metadata Model is based on the CWM OLAP package, which exposes business data in a multidimensional format that specifically supports data analysis. You use it to retrieve CWM-compliant metadata objects from an OLAP data source.

API Documentation:

Refer to the following Javadoc package for the OLAP Metadata Model's API documentation:

org.omg.cwm.analysis.olap

In the sections that follow, we first introduce some basic concepts of OLAP models, and then we highlight the corresponding CWM and SAP BI mappings together with extensions added on the BI Java SDK layer.

OLAP Systems

OLAP systems organize data typically drawn from multiple and diverse business information sources into a form that supports the fast analysis of this data to gain strategic business insight.

A primary characteristic of the OLAP model is that the data is presented in a multidimensional framework. This is a natural expression of the way business enterprises view their strategic data. Multidimensional models provide the data views needed to perform OLAP analyses, which look simultaneously at several dimensions of data, such as time, geography, and product.

Business analysts don't typically examine data in flat queries of two dimensions or less, but rather in more complex analysis such as "How much money in total returns were there in San Jose in Q1?" – a three-dimensional question. Complex queries can include six, seven, and even more dimensions. The OLAP engine of SAP's Business Information Warehouse (SAP BI), for example, executes queries that are formulated from such business questions. It allows you to quickly switch between different orientations of the dimensions, as well as between various structural arrangements of the data values of the dimensions.

The sections below introduce the basic components of the OLAP system. SAP BI users should note that BI occasionally deviates from this system, and in the BI Java SDK we base our OLAP Metadata Model upon the CWM OLAP package that more closely approximates the OLAP standard, while also adding some extensions to this standard. We talk about BI, CWM, and SDK mappings in the Mapping of OLAP Metadata section, below.

Cubes and Dimensions

The central object of a OLAP system is a cube. A cube is a logical organization of data for reports and analyses, organized as a collection of measures that share the same dimensionality.

Dimensions are sets of related identifiers or attributes of the data values of the system. They help categorize the data, or represent the attributes of the data values of the system.

The figure below illustrates a simple, three-dimensional OLAP cube:

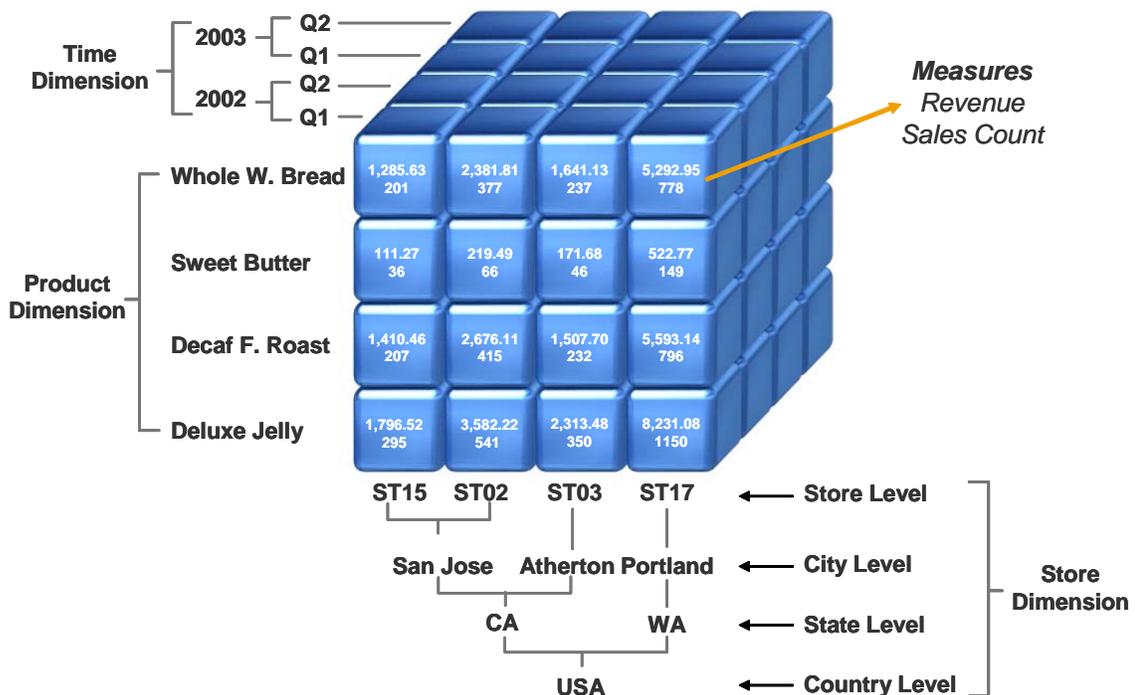


Figure 11 — OLAP Cube

Each of the three dimensions (Time, Store, and Product) consists of a set of related members. In this visualization, the edges are formed by the values of the different categories that form the key. The cells of the cube, formed at the intersection of coordinates of all categories (representing a particular value tuple), contain the measures.

A cell represents data values and their relevant attributes; for example, in referring to the figure above, **store** ST02 booked a certain revenue and sales count for the **product** Deluxe Jelly in the **timeframe** of Q1 2002. The revenue and sales count values are contained in the cell, and the cell is defined by the three different attributes:

- Store

Metadata APIs

- Product
- Time



Note:

- Not all cells contain data. For certain combinations of members, no data exists and the corresponding cell is said to be empty.
- Since the number of cells in a cube equals the product of the sizes of all dimensions, OLAP cubes can be very large.

Members and Properties

The individual elements of a dimension are called members. In the case above, the `Store` dimension has the members: `ST15`, `ST02`, `ST03`, `ST17`, `San Jose`, `Atherton`, `Portland`, `CA`, `WA`, and `USA`.

Members often carry additional descriptive information or dimension properties. A product may have properties like color, weight, and size, or a customer may have properties like country, city, and phone number.

Measures

Measures, or key figures in BI terminology, are the quantifiable values – such as currency amounts, in the above example – that are stored in a cube. Measures are typically of a numeric data type and often aggregate by simple summation. However, other data types such as date and time, and different aggregation behaviors such as average, minimum, or maximum are also possible.

For example, `Sales` would usually be a currency type, where `Quantity` is an integer, and `Weight` a real number. Sales values may be summed when aggregated over time, whereas average or final values are more adequate for inventory data.

For convenience and simplification, measures are often treated as members of another dimension of a cube – the measures dimension – which makes the model more symmetrical.

Hierarchies, Levels, and Default Members

In ODBO-specific data sources, the members of dimensions are usually arranged in a hierarchical structure, and sometimes there can be more than one “natural structure” or hierarchy. For example, think of large retailers or government agencies that have millions of customers. You may wish to report revenues of a set of customers grouped by countries and continents, and all continents finally roll up into a root node of the hierarchy. At different times, you want to look at the customers individually or as families arranged according to other criteria.

Levels are the different categories by which objects that share a number of common attributes in a hierarchy are grouped. The `Store` dimension in our example above, Figure 11 — OLAP Cube, has four levels:

- Country
- State
- City
- Store

A data value of a cube can only be unambiguously identified by specifying all its attributes, or its dimension values. In a cube with many dimensions, it is therefore convenient to use the concept of a default member, or ALL member. If the value of a dimension is not specified, a query processor can set its value to the default member. Typically, this is a member that represents the aggregate for all members of the dimension.

Member Selection Based on Level

Many hierarchies have at least two levels, the first of which is the ALL level. The ALL level usually contains just one member, the ALL member, which represents the aggregated values of all members.

It's important to note that it is often not meaningful to return the ALL member in a result set, but rather to return results based on a specific level, beginning with the second level, which begins the set of "real" members. Consider the hierarchy diagram below:

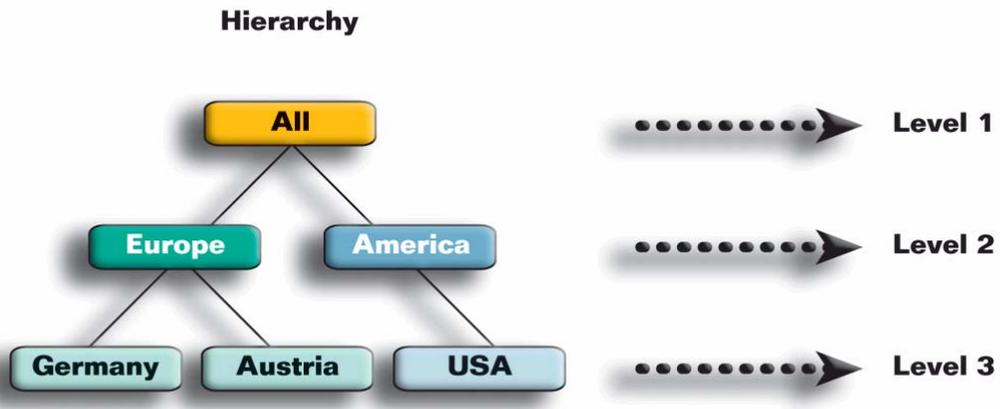


Figure 12 — Hierarchies and Levels

Metadata APIs

This is a geography hierarchy, with three levels shown:

- Level 1: ALL level
- Level 2: Region level
- Level 3: Country level

If you create a query that returns the top five countries based on sales without restricting to a specific level, the first result returned would be "All Countries," because the highest sales would be the total of all sales values across the hierarchy – the ALL member. The second two results would be "Europe" and "America." Instead, you would want to create a query based on a specific level – the Country level.



Example source code:

See the following Java code for an example of using the OLAP Command Processor to select the members of the right level for this query:

```
Level countryLevel = olap.getObjectFinder().
    findLevelFirst(geographyHierarchy, "Country");
commandProcessor.addLevelMembers(countryLevel);
commandProcessor.createTopCountFilter(geographyDimension, 5, salesMeasure);
```



Note

For the API documentation on the OLAP Command Processor, see the Javadoc at com.sap.ip.bi.sdk.dac.olap.query.IBICommandProcessor.

Query Operations

An OLAP API must provide operations that allow an application to rapidly navigate between different views of the data stored in a cube. The most common operations are:

- Pivot – Pivoting rearranges the projection of dimensions on the axes of a multidimensional result set.
- Drill up – Drill up operations navigate in a dimension to lesser detail.
- Drill down – Drill down operations navigate in a dimension from lesser detail to greater detail.

Mapping of OLAP Metadata

As we noted above, the SAP BI OLAP model occasionally deviates from the CWM OLAP model. This section describes how SAP BI objects are mapped to the objects of the CWM OLAP model, and particularly how CWM object names, and the names we use in the BI Java SDK, are derived from the corresponding identifiers used in SAP BI.

The class diagram below shows the major classes of the CWM OLAP model and their associations:

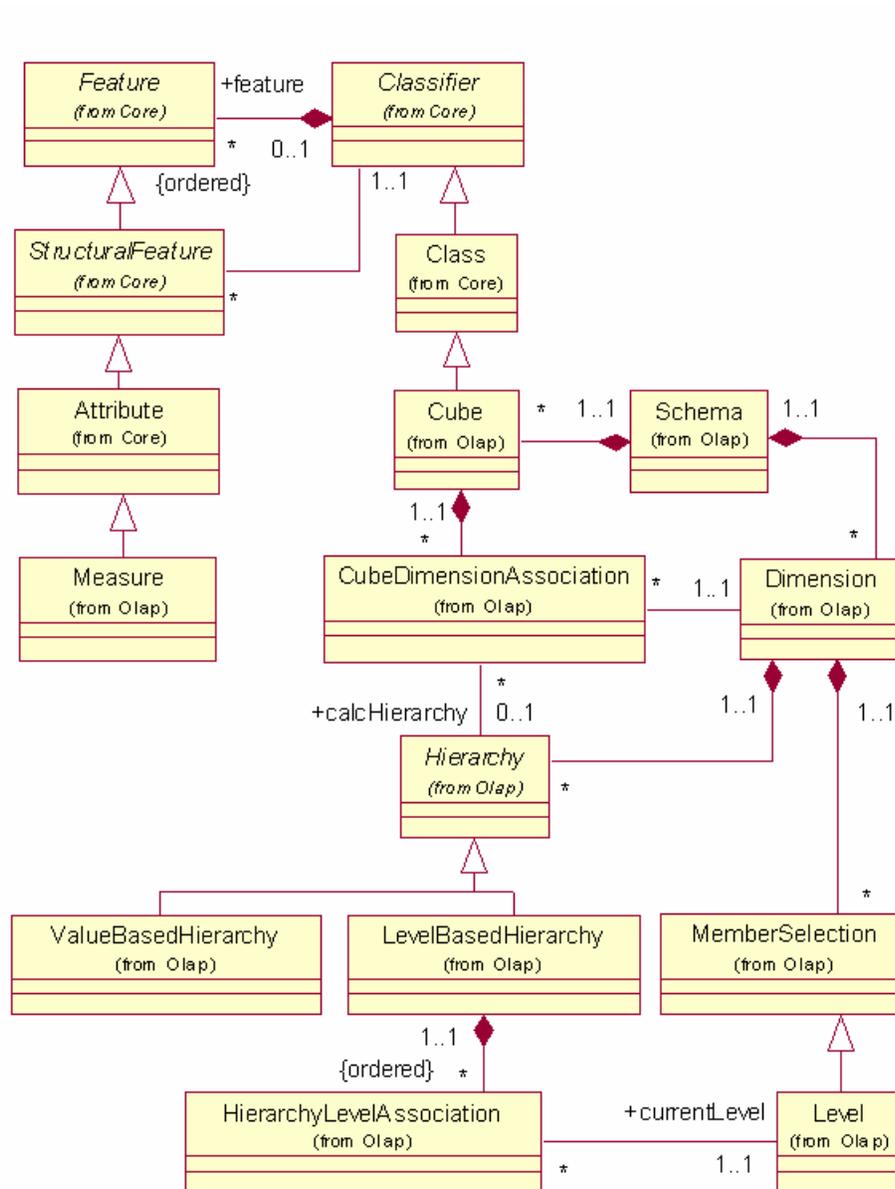


Figure 13 — CWM OLAP Model



Notes:

- All deployment-related classes, such as `CubeRegion`, `CubeDeployment` and `DimensionDeployment`, are omitted from the model. The methods to navigate the corresponding associations to these classes are not implemented by the SDK, and if used return empty collections.
- For a detailed description of each class, refer to the CWM Javadocs included in the BI Java SDK documentation set.

Introduction to the CWM OLAP Model

As depicted in the diagram above, the `Schema` class serves as a container for `Dimensions` and `Cubes`. A `Cube` is associated with a set of `Dimensions` and a set of `Measures`. `Dimensions` can have multiple hierarchies. There are two specializations of hierarchies: hierarchies that support ordering members by `Level` (`LevelBasedHierarchies`), and hierarchies that support ordering members by value (`ValueBasedHierarchies`). The `Dimension` class supports the identification of special dimensions by providing the boolean attributes `isMeasure` and `isTime`.

Measures and Attributes

If you are familiar with OLE DB for OLAP or XML for Analysis and the concept of schema rowsets, or if you are new to the CWM model, you may feel that certain key concepts such as measures and dimension attributes are missing in this model. These objects are provided in a more generalized form in a parent class. To illustrate, consider the relationship between a `Dimension` and its attributes. `Dimension` is a subclass of the class `Class` in the CWM Core package. The dimension's ability to have attributes is inherited from `Classifier` in its association with `StructuralFeatures`, of which `Attribute` is a specialization. The JMI interfaces rendered from this model allow retrieval of the attributes of a dimension with the `getFeature()` method. The objects returned can be cast into `Attribute`.

Likewise, in this model, measures are considered features or attributes of a cube. The cube has the same relationship to `Classifier` as `Dimension`, and `Measure` is a subclass of `Attribute`.

Members

Another distinction of the CWM OLAP metamodel is that there is no direct association between the `Dimension` class and a `Member` class. The JMI interfaces derived from the model therefore do not support any direct navigation – for instance, from a level to its members.

Metadata APIs

From a formal point of view, members are typically considered as data rather than metadata, and are therefore not within the scope of this metamodel. However, there are also good practical reasons to treat members differently. In contrast to metadata objects such as cubes, dimensions, hierarchies, and levels, members can occur in very large cardinalities. Exposing this information only through heavyweight JMI objects would be impractical when dealing with large dimensions such as products or customers.

Member information is therefore queried in the BI Java SDK either via full-blown queries, or with various convenience methods that are provided in addition to the JMI interfaces. These methods and the queries make member information accessible through a light-weight result set that is based on the JDBC ResultSet API.

Note that SDK OLAP Query API therefore provides an `IBIMember` class. This is required, since the query model relies on `Members` as metadata objects for input references in certain types of queries.

Mapping Table

The table below lists the major classes of the CWM OLAP metamodel (together with the BI Java SDK-specific extensions) and how they relate to corresponding SAP BI objects. The mappings described here are essentially the same mappings as the mappings used by the MDX-based APIs (OLE DB for OLAP, XMLA, and the OLAP BAPI). Since the BI Java SDK provides access to SAP BI via the BI XMLA and the BI ODBO Connectors, the mappings are determined to a large extent by the mappings that already occur on the XMLA interface.

Notes and additional details follow the table.

CWM OLAP Metamodel (and SDK extensions) and SAP BI Equivalents

CWM OLAP metamodel and SDK extensions	SAP BI equivalents	Mapping notes	CWM object in SAP BI name format (used in the SDK)
Schema	N/A	Since <code>Schema</code> is not an optional element of the CWM model, one <code>Schema</code> object with the name <code>\$INFOCUBE</code> is created as a container for all <code>InfoCubes</code> of a system. In addition, there is one <code>Schema</code> for each <code>InfoCube</code> that contains a query that is released for the OLE DB for OLAP (ODBO) interface.	<code>\$INFOCUBE</code> <InfoCube> Examples: <code>\$INFOCUBE</code> <code>0D_SD_C03</code>
Cube	InfoCube, Query	In addition to <code>InfoCubes</code> , BEx queries that are released for external access via the ODBO interface are also mapped to <code>Cube</code> objects.	<code>\$<InfoCube></code> <InfoCube/ technical query name> Examples: <code>\$0D_SD_C03</code> <code>0D_SD_C03/0D_</code> <code>SD_C03_Q009</code>
Dimension	Characteristic,	The dimensions for a cube are filled	<InfoObject>

CWM OLAP metamodel and SDK extensions	SAP BI equivalents	Mapping notes	CWM object in SAP BI name format (used in the SDK)
	Structure	with all free characteristics and the special measures dimension. In addition, SAP BI structures are mapped to dimensions. In the BEx Query Designer, you can set a technical name in the properties of structures or structural components. Otherwise, the UNIQUE-ID (UID) is used.	Examples: 0D_DIV 0D_SALE_ORG 0CALMONTH
Hierarchy	Hierarchy	In SAP BI, each dimension has a default hierarchy with the same name as the dimension to which it belongs (first example listed). The second example listed is an example of an external hierarchy exposed by the ODBO interface.	<InfoObject or hierarchy name> Examples: 0D_DIV 0HYEA1_MON
Level	N/A	Hierarchies in SAP BI are not leveled in a strict sense. The ODBO interface constructs level names by numbering the levels.	LEVEL## (Where ## is a two-digit number ranging from 00 to 99.) Examples: LEVEL00 LEVEL01
Measure	Key Figure, Structure Element	Measures are handled in the SDK as Members (type IBIMember) of a special Dimension, the Measures dimension. In the SDK, measures are also accessible as features of the cube, of type Measure. Call Dimension.isMeasure():boolean to determine whether a dimension is a measures dimension.	<InfoObject> Examples: 0D_COSTVALS 0D_OORVALSC
Attribute	Attribute		[<InfoObject>] (Note that the surrounding angle brackets are required in the SDK naming convention.) Examples: [10D_COUNTRY] [10D_DIV]

CWM OLAP metamodel and SDK extensions	SAP BI equivalents	Mapping notes	CWM object in SAP BI name format (used in the SDK)
Member (SDK extension)	Characteristic value	In the SDK, members are associated with a dimension. <code>IBIMember</code> is an extension added on the SDK level.	<characteristic value> Examples: 7 DE All
SAP variable (SDK extension)	SAP variable	<code>IBISapVariable</code> is one of two extensions added on the SDK layer to support SAP Variables. We've also extended <code>Cube</code> with <code>IBICubeOwnsSapVariable</code> , <code>IBISapVariable</code> 's association with <code>Cube</code> .	[<SAP variable>] (Note that the surrounding angle brackets are required in the SDK naming convention.) Examples: [0D_SA_OR] [0D_DISCH]



Notes:

- The elements in brackets in the naming column, such as <InfoCube>, indicate that the name of the CWM object is formed by the name of the corresponding SAP BI object. For example, where the name of the InfoCube is 0D_SD_C03, the name of the CWM cube is \$0D_SD_C03. Note also that to refer to a BI cube, you must prefix its name with "\$".
- The naming examples in the table above are taken from the SAP Sales DemoCube: Overview (see its documentation on the [SAP Help Portal](#)). The OLAP tutorials and examples are also based on this cube. See in particular OLAP 1 - Accessing OLAP metadata, for additional examples of working with metadata in the BI Java SDK. To view the complete list of examples, start at the root of the documentation set ([index.html](#) in the docs folder of the unpackaged distribution archive) and choose Examples.
- In the SDK, we recognize the name value of the InfoObjects (not the unique name or technical name).

Schema and Cube

The schema concept in the CWM specification is not supported by SAP BI. The current version of SAP's OLE DB for OLAP implementation allows for various options to access the data stored in an InfoProvider:

- Direct access to the data of an InfoProvider:

With certain restrictions, you can directly access data belonging to a BI InfoProvider, using the MDX command `$InfoProvider`. This option is available for the following objects:

- for all InfoCubes having type `BasisCube` and for MultiProviders as `Cubes`

Metadata APIs

- for all characteristics and key figures as *Dimensions/Measures*

Exceptions are navigation attributes and key figures that are neither restricted nor calculated.

- Access to the data of an InfoProvider using a query:

In this approach, you use the BEx Query Designer to define queries for the requested InfoProvider, and use these queries as data sources. This is the recommended approach.

The following diagram illustrates the mapping of a BEx query onto a cube:

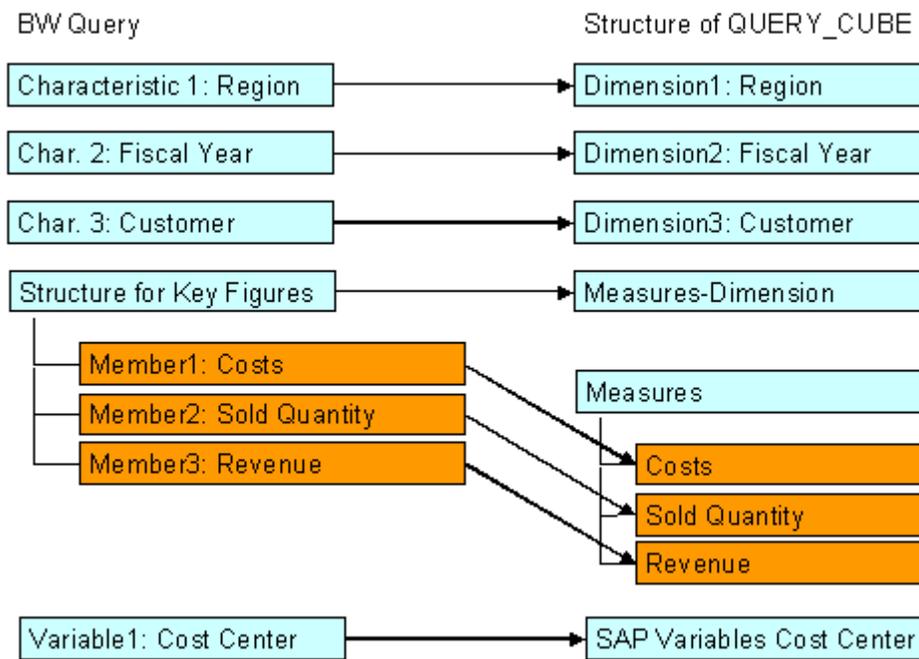


Figure 14 — BEx Query Mapping

Measures

SAP BI's calculated and restricted key figures are mapped to measures by default. This gives the BI Java SDK API consumers access to complex key figure definitions. Defining calculated and restricted key figures on the provider (SAP BI) side of the interface bears the following advantages:

- Potential performance optimizations on the SAP BI side
- Less complexity for the client application

Metadata APIs

- Consistent interpretation and use of derived key figures guaranteed for various application areas.

Hierarchies and Levels

Every characteristic in a SAP BI query is modeled to a dimension with a flat hierarchy. The name of this hierarchy is the same as the name of the dimension. This hierarchy has the following levels:

- Level 0 with ALL members (all members are displayed regardless of their position in the hierarchy).
- Level 1 with a subset of master data table values.

SAP BI hierarchies can be established as additional hierarchies. The default hierarchy for a dimension is filled with the presentation hierarchy of a BI query definition.



Note:

For additional information on metadata mapping and SAP BI, see the [Mapping the Metadata](#) document on the SAP Help Portal.

Relational Metadata Model

The Relational Metadata Model is based on the CWM Relational package and describes data accessible through a relational interface such as JDBC. You use it to retrieve CWM-compliant metadata objects from a relational data source.

API Documentation:

Refer to the following Javadoc package for the Relational Metadata Model's API documentation:

org.omg.cwm.resource.relational

In the sections that follow, we first introduce some basic concepts of relational models, and then we highlight the corresponding CWM mappings.

Relational Databases

Relational databases are repositories for typically large amounts of information, structured in accordance with the relational model, in tables with columns. They are created and administered by relational database management systems (RDBMSs). With an RDBMS, you define storage structures for data and mechanisms for its manipulation and

Metadata APIs

retrieval. An RDBMS must also provide a system for safeguarding in case of events such as system crashes and unauthorized access.

The relative ease with which data can be managed with relational databases is one of the main factors for the success of the RDBMS. Data definition and manipulation is typically done with the help of the Structured Query Language (SQL), which is supported to various degrees by the vast majority of currently available RDBMSs. Application development that relies upon access to relational databases is simplified by standard APIs such as Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC).

The relational database typically consists of physical storage spaces also known as catalogs, which in turn contain a set of namespaces known as schemas. The schemas contain the tables which hold the data. Since the tables are organized as sets of rows with an identical structure per table, they represent relations - hence the name relational databases. The rows of tables represent tuples in a mathematical sense. The mathematical theory behind the relational model has been thoroughly investigated and has led to the design of database languages such as SQL and their underlying execution model.

Tables

The central object of the relational model is the relation, also known as the table. A table is a set of rows (or tuples). The term “set” in this context is mathematical, since most tables have an associated unique index that ensures the set property. The tuples of the table each possess an identical structure, with columns that have the same name and data types and possible additional information such as participation in indexes.

Rows

The rows (or tuples) in a table contain the data of the table. Each row has the same structure, predefined for a particular table.

Columns

Columns describe the structure and type of the rows of a table. They have a name, a data type, and an implicit order (ordinal) based on the order chosen when defining the table. Columns can also belong to indexes, which are used to ensure uniqueness (set property) of rows in the table.

For example, if columns $c_1 \dots c_n$ of table t belong to a unique index i , then no two rows r_1 and r_2 can exist in t , with $r_1(c_1) = r_2(c_1), \dots, r_1(c_n) = r_2(c_n)$.

Relational Query Operations

Queries in the relational model can be formulated with the help of a set of operators of the relational algebra. These operators have direct representations in all relational query languages that have the same capability of expression as the relational model, which includes SQL. Each operator returns a relation as the result of its execution, and this relation supports the construction of complex queries by nesting these operators.

A typical SQL statement is semantically equivalent to a set of nested operators of the relational algebra. Most relational databases use an internal representation of SQL statements similar to the relational algebra. This facilitates the optimization of queries, since relational algebra supports the concept of so-called equivalence transformations, or transformations that do not change the result of the expression. For example, `select (project)` is equivalent to `project (select)`, however, the latter is more efficient since the projection only takes place on the result of the selection, which is in general smaller than the relation upon which it operates.

The list of operators, and their SQL equivalents, is described in the table below.

Relational Query Operators and SQL Equivalents

Operator	Description	SQL equivalent / notes
<i>select</i>	Takes a relation and a predicate (logical expression) and allows restriction of the relation to the rows that satisfy the given set returned as the result of a query.	WHERE
<i>project</i>	Takes a relation and a set of columns and returns a new relation containing all rows of the given relation where each row of the result contains only those columns contained in the given set of columns.	SELECT
<i>rename</i>	Takes a relation and either a name or a map of column names to new column names. In the first case, <i>rename</i> renames the relation itself. This case functions to support multiple uses of the same relation in a complex query – or self-joins. The second case supports the renaming of columns in the result, and is used in Cartesian products involving tables that have identically named columns.	In SQL, the FROM and SELECT clause support renaming of columns and tables which thereafter can be referred to using their new names, for example in the WHERE clause.
<i>Cartesian product</i>	Takes two relations and returns a relation containing the Cartesian product of both, given relations where each row is a tuple comprised of the columns contained in both given relations. In other words, for each possible pair of rows of both relations, the result contains a row comprised of the concatenation of the rows of that pair.	FROM clauses in SQL containing more than one table specify the Cartesian product of all tables involved.
<i>union</i>	Takes two relations of identical structure (with the same number and data type of columns) and returns the set comprised of all the rows of both given tables.	UNION
<i>set difference</i>	Takes two relations of identical structure (with the same number and data type of columns) and returns the set comprised of all the tuples of the first relation that are not contained in the second relation.	INTERSECTION

Most relational query languages extend the relational algebra to support operators that allow the manipulation of the result in a way that can not be represented in the relational algebra, adding additional operators such as:

Additional Operators and SQL Equivalents

Operator	Description	SQL equivalent / notes
sort	Takes a relation and a set of columns with sort direction (ascending, descending) and returns the relation with the rows ordered accordingly.	ORDER BY
aggregate	Takes a relation and a set of columns with aggregation functions. In addition, each aggregation function can be accompanied by a set of columns by which the aggregation is to be grouped — one aggregated value of the column for each distinct member of the group. The result is a set of rows with the columns aggregated according to their chosen function (for example, <code>sum</code> , <code>min</code> , <code>max</code> , <code>avg</code> , <code>count</code>) and grouped by the columns chosen for the aggregation function.	SQL supports this in the <code>SELECT</code> clause with aggregation functions on the selected columns and using the <code>GROUP BY</code> clause to specify the grouping.

Mapping of Relational Metadata

The CWM Metamodel consists of 21 separate packages. One of these is the Relational package, which can be used to describe SQL99-compliant relational database schemata. Nearly all current RDBMS implementations support SQL99 or subsets of SQL99, which means that CWM’s Relational package can be used to support CWM-based metadata interchange between these systems.

The following class diagram shows the subset of the classes used in the BI Java SDK that make up the CWM Relational package:

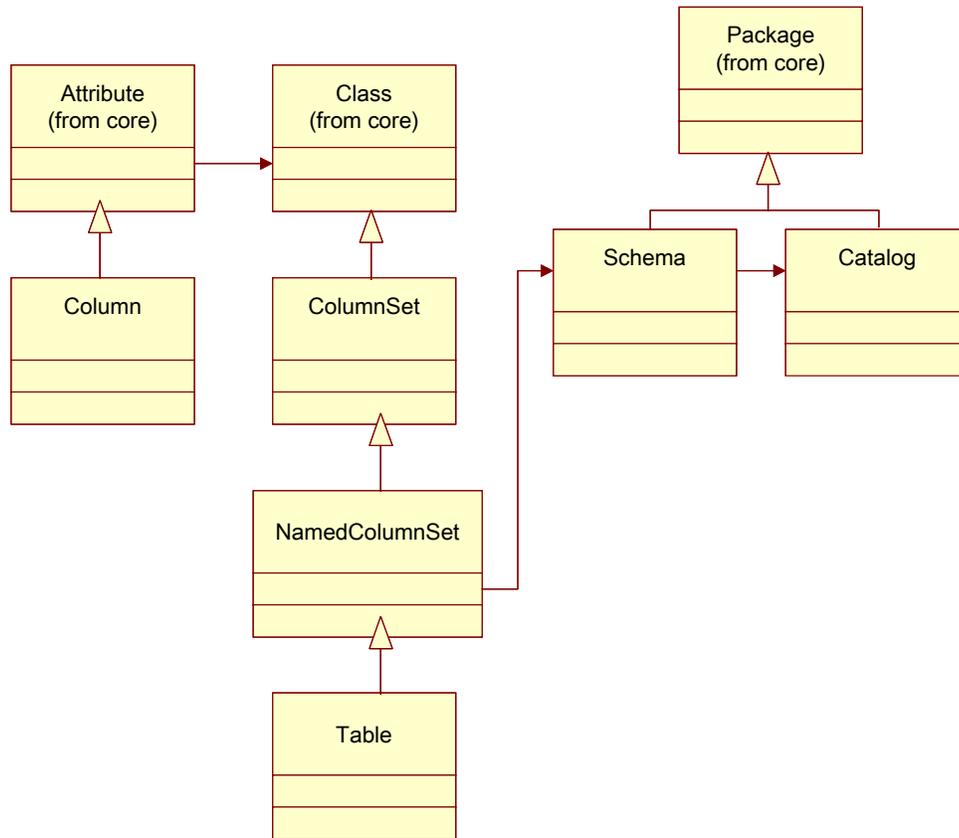


Figure 15 — CWM Relational Package

As expressed in the diagram, the BI Java SDK uses the following subset of classes and their relationships from the CWM Relational package:

- Catalog
- Schema
- Table
- Column

The Relational package uses other packages of the CWM Metamodel, evident in the diagram via inheritance, to provide necessary functionality without having to repeatedly design similar concepts in the various packages. For example, Table is a subclass of Class, and Column is a subclass of Attribute. Each instance of Class owns an ordered list of instances of Attribute via the ClassifierFeature association. Therefore, the instances of Column that belong to a Table instance can be retrieved via the same ClassifierFeature association.

Examples

The JMI interfaces rendered from the Relational package allow us therefore to navigate from a given `Table` instance to its `Column` instances via the `getFeature()` method, which returns a list of `Attribute` instances that can be cast into instances of `Column`.

Examples

Although most of the SDK's examples retrieve metadata from data sources, the following three examples focus in particular on accessing metadata:



Olap_1.java – Accessing OLAP metadata:

Demonstrates four different ways to retrieve OLAP metadata:

1. Via connection-level methods:

The connection-level methods `getSchema()`, `getCube()`, `getMemberData()`, and `getTaggedValue()` allow you to browse top-level metadata objects such as cubes and schema. Typically, the objects retrieved are then used as an entry point to further "navigate" to objects that are contained in these name-space-like objects.



Caution:

Be careful when retrieving data from cubes and consider displaying data for only one cube at a time. The results could be too large and could impact performance if there are many cubes. In this example, we only display the data for the first cube.

2. Via `ObjectFinder` methods:

`ObjectFinder` methods provide the ability to retrieve a specific object or set of objects. Note that there are always four find methods for each object type (such as `Cube`, `Dimension`, and `Hierarchy`) that differ in their signatures:

- i. `finder.findDimension(Cube cube, String dimensionName)`
- ii. `finder.findDimensionFirst(Cube cube, String dimensionName)`
- iii. `finder.findDimension(String schemaName, String cubeName, String dimensionName)`
- iv. `finder.findDimensionFirst(String schemaName, String cubeName, String dimensionName)`

The first two methods use a `Cube` object and a `dimensionName` to identify the dimension(s) to be found. The `findDimensionFirst()` methods are simply for convenience; they are equivalent to `(Dimension) finder.findDimension(cube, dimensionName).get(0)`.

Examples

3. Via CWM-based JMI interfaces:

Starting from the top level objects of the OLAP Metadata Model, such as `Cube`, you can use the JMI interfaces provided by each object to "navigate" to associated objects. The example in this section shows how to retrieve from a cube its associated dimensions, the hierarchies of a given dimension, and the levels of a given hierarchy.

Note that in the OLAP package of the CWM metamodel, members are not directly associated with levels. This is rooted in the considerably different nature of members compared to other objects such as cubes, dimensions, hierarchies and levels. Members somehow straddle the line between data and metadata, and in particular, members can potentially occur in very large cardinalities. For example, a customer dimension of a large retail data warehouse may have millions of entries. Heavy weight metadata objects are thus not suitable to represent members.

4. Via member data access methods (`getMemberData(List, List)`)



Relational_1.java – Accessing Relational metadata:

Illustrates the process of retrieving relational metadata from catalog to column from a JDBC data source.



Relational_2.java:

Demonstrates three different ways to retrieve relational metadata:

1. Via connection-level methods:

The connection-level methods `getCatalog()`, `getSchema()`, and `getTable()` allow you to browse metadata objects such as schemas and tables. Typically, the retrieved objects are used then as an entry point to further "navigate" to objects that are contained in these name-space-like objects.

2. Via `ObjectFinder` methods:

`ObjectFinder` methods provide the ability to retrieve a specific object or a set of objects.

3. Via CWM-based JMI interfaces:

Starting from the top-level objects of the Relational Metadata Model, such as `Table`, you can use the JMI interfaces provided by each object to "navigate" to associated objects. The example in this section shows how to retrieve from a given table its associated columns and the schema and catalog to which it belongs.



Note:

See Appendix B: Examples for the full index of examples and instructions on getting your system up and running with them.

Chapter 4: Creating Queries

Overview

Build queries based on the metadata in the SDK's CWM-based metadata models using the SDK's query APIs. These APIs provide methods to create and execute complex OLAP and relational queries.

This chapter describes the SDK's query APIs and their underlying models in the following sections:

- Query APIs
 - OLAP Query Model
 - Relational Query Model

API Documentation:

See the individual subsections.

Query APIs

The SDK provides two query APIs, both generated via JMI (see Generating Interfaces, above) from their respective query models:

- OLAP Query API, generated from the OLAP Query Model, for defining queries against an OLAP server
- Relational Query API, generated from the Relational Query Model, for defining queries upon relational data sources

To support the definition of often complex queries in a methodical, step-by-step process, the query APIs also include simplified command processors. These are interfaces that are part of the query APIs and make it easier to use them by hiding the complexity of the underlying query models. With the command processors, you can create and manipulate complex queries with simple commands. The SDK provides two command processors:

- OLAP Command Processor
- Relational Command Processor

Query APIs

Most of the time, the command processors will be all you need to query data sources. However, to leverage the full functionality of the query APIs, you will want to understand their underlying query models. To that end, the rest of this section discusses the query models in more detail.

For additional documentation:

- To learn how to use the methods of the OLAP or Relational Command Processor, refer to the Javadocs for the respective `IBICommandProcessor` class (see table, below)
- For step-by-step examples of using the query APIs and command processors, refer to the examples that ship with the SDK. See a listing in the Examples section at the end of this chapter.

Components of the SDK's Query APIs:

Component	OLAP	Relational
Command Processor	com.sap.ip.bi.sdk: dac.olap.query.IBICommandProcessor	com.sap.ip.bi.sdk: dac.relational.query.IBICommandProcessor
Query API	com.sap.ip.bi.sdk.dac.olap.query.*	com.sap.ip.bi.sdk.dac.relational.query.*
Generated from	OLAP Query Model	Relational Query Model

OLAP Query Model

The OLAP Query Model provides the building blocks for creating OLAP queries. The model is an abstract and source system-independent way to describe a multidimensional (OLAP) query and therefore specify a multidimensional result set. This allows you to formulate OLAP queries based on the CWM-compliant metadata provided by the SDK's Metadata APIs independently of data source-specific APIs.

The model consists of various components, many of which are expressed with UML diagrams in this section and in their respective Javadocs packages. Although the OLAP Query Model has been designed specifically for the SDK, it relies on several CWM packages as well.



Note:

If you are viewing this guide with a color-capable display, the coloring used in the diagrams helps you determine the origin of the object. Light yellow represents objects or classes inherited from CWM. Light blue represents SDK-native objects or classes. A clear color represents an SDK-native object or class as well, but designates that it is abstract.

The below table summarizes the query model's components and associated SDK and CWM packages:

OLAP Query Model: components and packages

Component	Associated SDK package
OLAP Command Processor Query factories	com.sap.ip.bi.sdk.dac.olap.query
Main model AxisDimension	com.sap.ip.bi.sdk.dac.olap.query.main
Contains all associations between objects of the model	com.sap.ip.bi.sdk.dac.olap.query.assoc
InputReference Adapter classes NumericValueFunction Operation	com.sap.ip.bi.sdk.dac.olap.query.input
Member MemberExpressions	com.sap.ip.bi.sdk.dac.olap.query.member
MemberSetExpressions	com.sap.ip.bi.sdk.dac.olap.query.msx
TupleSetExpressions	com.sap.ip.bi.sdk.dac.olap.query.tsx
Types	com.sap.ip.bi.sdk.dac.olap.query.types
SAP variables support	com.sap.ip.bi.sdk.dac.olap.query.var
Component	Associated CWM package
OLAP Metadata Model	org.omg.cwm.analysis.olap
Foundation for CWM metamodel development	org.omg.cwm.objectmodel.core
Provides the Member Object	org.omg.cwm.resource.multidimensional
Provides Object, from which Member is derived	org.omg.cwm.objectmodel.instance

For API documentation, refer to the Javadocs for these packages via the links in the table above, or start with the Javadocs overview pages:

- [SDK Javadocs overview page](#)
- [CWM Javadocs overview page](#)

Basics of OLAP Queries

We can illustrate the components of an OLAP query beginning with the following representation of its result set:

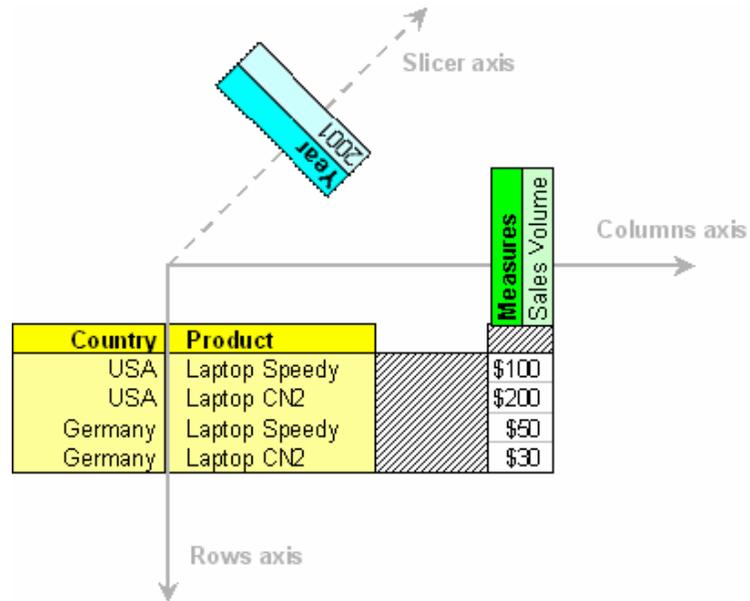


Figure 16 — The OLAP Result Set

A multidimensional result set is composed of a set of axes – rows, columns, and a slicer, in the above example. There can be as many axes as desired, but it’s easiest on paper to represent rows and columns, with the slicer on the additional “third dimension.” Each axis is populated with tuples. In the case above, two dimensions – Country and Product – have been assigned to the rows axis, and the rows axis is populated with tuples belonging to both these dimensions:

- USA, Laptop Speedy
- USA, Laptop CN2
- Germany, Laptop Speedy
- Germany, Laptop CN2

Both the columns and the slicer axes have exactly one member selected (Sales Volume and 2001), and are populated with the tuples those members describe.

The intersection of each of the tuples from all axes forms a cell value. These cell values can be regarded as the numbers the query returns, which in the above case are:

- \$100
- \$200

Query APIs

- \$50
- \$30

In other words, \$100 of Speedy Laptops were sold in the USA in 2001, \$200 of CN2 Laptops were sold in the USA in 2001, and so on.

This is a simplified representation, but you can see that when you add additional axes and populate your axes with additional dimensions, you can construct highly complex queries.

Main model

In order to build queries that specify results such as the above, we begin defining the OLAP query model with the Main Model. The Main Model contains the major OLAP query objects, many of which in turn form the top-most objects of subsequent sub-models.

The Main Model is diagrammed below:

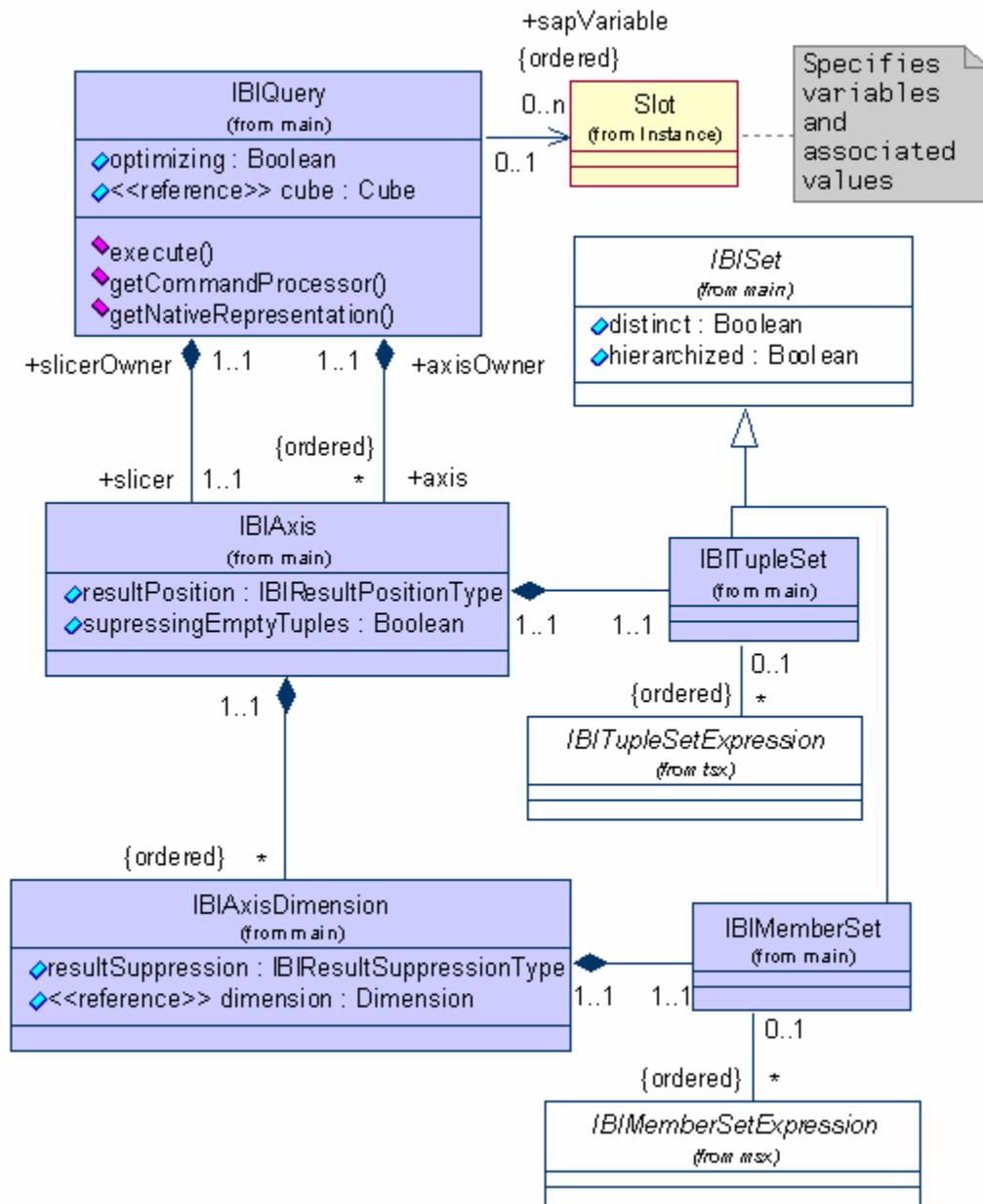


Figure 17 — Main Model Diagram

The query model is composed of many classes. At its root in the Main Model is the `IBIQuery` class, which can be regarded as the top most element of a query.

Query and Axes

A multidimensional query is accomplished by distributing dimensions of a cube on different axes, hence a query aggregates a number of axes. There are two different types of axes belonging to a query: regular axes, and a special slicer axis. Correspondingly, note that in the diagram above, `IBIQuery` has two associations with `IBIAxis`: the slicer association and the axis association.

There is a one-to-one relationship between a query and its slicer axis: a query always has one, and only one slicer. However, the query can have zero to many regular axes. As you can see from the diagram, `IBIQuery` therefore aggregates exactly one slicer axis, and zero to many regular axes.

The main difference between the two sets of axes is that the result set for the query will have axes specifications for the non-slicer axes, but none for the slicer axis.

The Slicer

The slicer axis functions like a filter, influencing only the cell values. The query is filtered by all members of the various dimensions populated on the slicer axis. For example, you may want to restrict all sales values returned in a result set to a certain year, in which case you would have the slicer axis define a particular year by selecting only the member for that year.

The concept of a slicer can be illustrated in two different ways:

- From the point of view of query tool: OLAP tools typically provide a facility to page through different subsets of data, for example, to sort through certain calendar years or product groups. The value you specify for the slicer axis, calendar year in this case, allows you to do that.
- In terms of MDX equivalents: slicer values are the values that go into the `WHERE` clause of an MDX statement.

The slicer axis is sometimes referred to as the page axis or the filter axis.

A slicer axis is not displayed in the visual sense, it simply restricts or filters the values in the result set. On the other hand, the regular axes contribute directly to the result set display. There are only two that can be easily displayed in two dimensions: column and row. However, a query can have an unlimited number of axes to it.

The axes therefore define the geometry of the query by orienting the dimensions of the cube along them. Typically, a tool would display three axes: the slicer (to specify filter values and a subset of data of cube), and columns and rows (the two visible axes), which allows the display of data in tabular, spreadsheet-like fashion. Remember that the query itself, however, is not limited to three axes.

The Cube

A query is based on a cube. The `Cube` object comes from the CWM OLAP package, and is the main part of a query, referenced by `IBIQuery`, and aggregating its axes. There is a many-to-one relationship between the query and its cube: one query is based on a one cube, selecting data from that one cube, though a cube can have many queries.

`IBIAxisDimensions` need to exist and be assigned to axes for all existing dimensions of the referenced cube.

Axis Dimension

An axis can have zero to many ordered `IBIAxisDimensions` assigned to it, where an `AxisDimension` is an ordered collection (a subset) of members of a specific dimension. The axis dimension is therefore the specification of selected members which have a certain order, aggregating a subset of members of the dimension that it represents.

`AxisDimension` and its related classes are further diagrammed below:

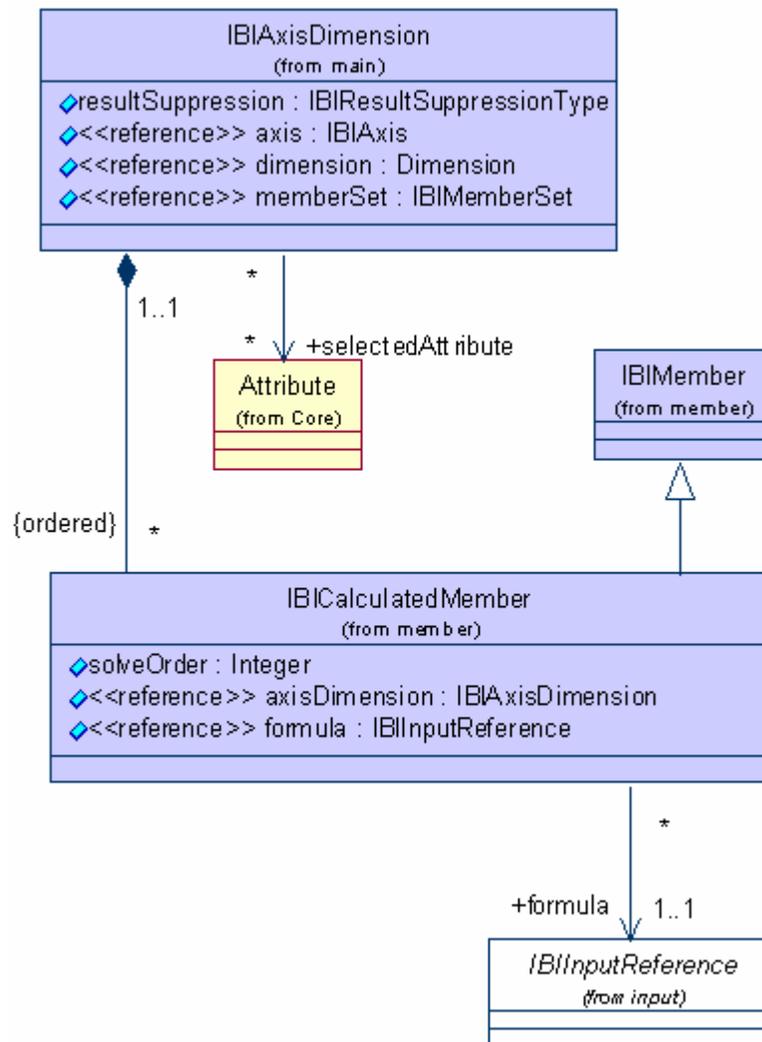


Figure 18 — AxisDimension Diagram

There are two ways to populate tuples on axes. One way is to directly specify the ordered collection of tuples – `IBITupleSet` – for an `IBIAxis`. An additional way is to specify member collections – `IBIMemberSets` – for the `IBIAxisDimensions` of an `IBIAxis`. The specified members of different `IBIAxisDimensions` of one `IBIAxis` will then be crossjoined, which is what builds tuples.

Both ways can be used at the same time. The crossjoin is evaluated first, and then the tuple set specified by `IBITupleSet` is joined by union.

Query APIs



Example:

Let's consider each way and demonstrate how to come to the same result with the two different approaches. As in our query example above, we assign the `IBIAxisDimensions` for two dimensions – Country and Product – to an `IBIAxis`. The query result set has the following four tuples populated on this particular `IBIAxis`:

(Country, Product):

- (USA, Laptop Speedy)
- (USA, Laptop CN2)
- (Germany, Laptop Speedy)
- (Germany, Laptop CN2)

1) Direct `IBITupleSet` specification

Exactly one instance of `IBITupleSet` is aggregated by an `IBIAxis`. This instance specifies the collection of tuples by having an ordered zero-to-many association with `IBITupleSetExpression`:

```
- IBIAxis
  - IBITupleSet
    - IBITupleList (sub-class of IBITupleSetExpression):
      (USA, Laptop Speedy)
      (USA, Laptop CN2)
      (Germany, Laptop Speedy)
      (Germany, Laptop CN2)
```

2) `IBIMemberSet` specification

Exactly one instance of `IBIMemberSet` is aggregated by an `IBIAxisDimension`. This instance specifies the collection of members by having an ordered zero-to-many association with `IBIMemberSetExpression`:

Query APIs

- IBIAxis
 - IBIAxisDimension (for Dimension Country)
 - IBIMemberSet:
 - IBIMemberList (sub-class of IBIMemberSetExpression):
 - USA
 - Germany
 - IBIAxisDimension (for Dimension Product)
 - IBIMemberSet:
 - IBIMemberList (sub-class of IBIMemberSetExpression):
 - Laptop Speedy
 - Laptop CN2

The resulting tuples are calculated by crossjoining the collections of members of the two IBIAxisDimensions:

USA	⊗	Laptop Speedy	⇒	USA, Laptop Speedy
Germany		Laptop CN2		USA, Laptop CN2
				Germany, Laptop Speedy
				Germany, Laptop CN2

TupleSetExpressions and MemberSetExpressions

`TupleSetExpressions` and `MemberSetExpressions` both specify an ordered collection of tuples and are derived from the abstract supertype `SetExpression`. The difference between the two is that an `IBITupleSetExpression` specifies a collection of tuples of members of one to many dimensions (with one member per dimension), while an `IBIMemberSetExpression` specifies a collection of tuples of members of one dimension (with one member per dimension). `MemberSetExpression` is therefore a specification of an ordered collection of members of one dimension.

Let's consider the classes shared by both together in one diagram:

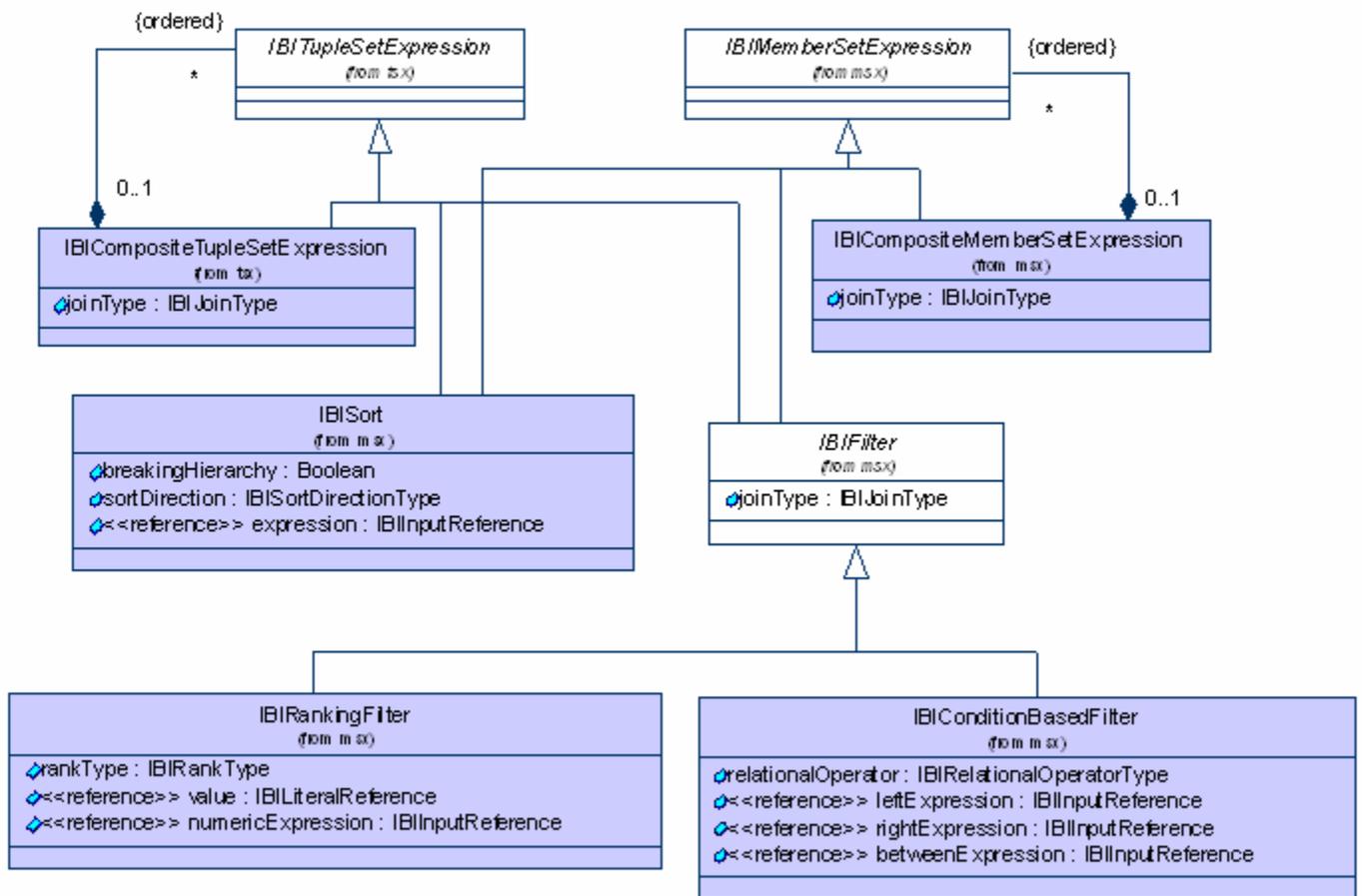


Figure 19 — TupleSetExpression and MemberSetExpression Diagram

The `IBITupleSetExpression` and `IBIMemberSetExpression` classes are abstract and can therefore not be instantiated. A list of subtypes are provided that can be instantiated in order to specify tuple collections in various ways (see “All Known Subinterfaces” in the [IBIMemberSetExpression](#) and [IBITupleSetExpression](#) Javadocs).

Regarding the existing subtypes of `TupleSetExpression` and `MemberSetExpression`, there are types which inherit only from `TupleSetExpression`, types which inherit only from `MemberSetExpression`, and types which inherit from both supertypes. The subtypes which are derived from both supertypes can be used to specify either tuple collections or member collections, depending on their parameterization.

The subtypes are `IBIFilter` (with the non-abstract subtypes `IBIRankingFilter` and `IBIConditionBasedFilter`) and `IBISort`, which can work on both tuple and member collections.

Categories of SetExpressions

Relevant for both `TupleSetExpressions` and `MemberSetExpressions`, there are four categories of `SetExpressions`:

1. **Select:**
 These expressions select tuples or members and specify how they should interact with tuples and members in preceding expressions:
 - **Members:** `IBICompositeMemberSetExpression`, and all subinterfaces of `IBIMemberSelection`.
 - **Tuples:** `IBICompositeTupleSetExpression`, and all subinterfaces of `IBITupleSelection` (which is only `IBITupleList`).
2. **Filter:**
 These expressions eliminate tuples or members from a given set:
 - `IBIRankingFilter` and `IBIConditionBasedFilter`
3. **Sort:**
 This expression changes the order of selected tuples or members:
 - `IBISort`
4. **Drill:**
 These expressions either expand or collapse nodes in a hierarchy:
 - **Members:** `IBILevelDrill` and `IBIMemberDrill`
 - **Tuples:** `IBITupleDrill`

Join Types

In order to specify how two tuple or member collections should interact with each other, the model declares six different join types:

Join types

Type	Description	Usage
APPEND	The resulting collection of tuples/members is computed by appending the current collection to the previously defined collection.	Regarded as the default join type, and used to gradually define the resulting collection. Potential duplicate members are retained.
INITIAL	The resulting collection of tuples/members equals the current collection. All previously-defined selections are ignored.	This is mainly used as join type for the first <code>TupleSetExpression</code> or <code>MemberSetExpression</code> in a sequence.
EXCEPT	The resulting collection of tuples/members equals the previously-defined collection, with all tuples/members of the current collection which also exist in the previously-defined collection removed.	This is used to build business questions like “all products that are blue, but not sold in Texas.”

Type	Description	Usage
GENERATE	For each tuple/member of the previously-defined collection, the current collection will be applied and a new collection created. The resulting collection of tuples/members is computed by joining the newly-created collections.	Only used if the current collection contains a variable like CURRENTMEMBER. With it, you can create the types of asymmetric result sets found in business questions like “the best 5 products of my best 5 stores.”
INTERSECT	The resulting collection of tuples/members equals the intersection of the previously-defined collection and the current collection.	Used to build business questions like “all products that are green and liquid.”
UNION	The resulting collection of tuples/members is computed by joining the previously-defined collection with the current collection.	Used to gradually define the resulting collection, like APPEND, but duplicate members are eliminated. Builds business questions such as “all products that are blue or liquid.”

Composite Design Pattern

You can define `IBIMemberSetExpressions` and `IBITupleSetExpressions` either in a sequence, or nested in a tree-like structure. Use parenthesis to specify more complex tuple/member collections, for example:

```
( blue ^ solid ) v ( green ^ liquid )
```

The `IBIMemberSetExpression` and `IBITupleSetExpression` models employ the concept of a *design pattern*, in this case, the composite design pattern, that specifies a treelike structure:

- Abstract component
 - Concrete composite
 - Concrete component

`IBIMemberSetExpression` and `IBITupleSetExpression` are abstract components. `IBICompositeMemberSetExpression` and `IBICompositeTupleSetExpression` are concrete composites, and all the non-abstract subtypes are concrete components (see “All Known Subinterfaces” in the [IBIMemberSetExpression](#) and [IBITupleSetExpression](#) Javadocs).

The composite design pattern is applied twice: once on the level of `IBIMemberSetExpressions`, and once on `IBITupleSetExpressions`. The exception is that there are classes which are concrete components of both design patterns: `IBIFilter` and `IBISort`.



Note:

For more information on the composite design pattern, see Gamma, Erich; et. al. *Design Patterns*. Refer to the Appendix for the reference.

Query APIs

The following example for an `IBICompositeMemberSetExpression` shows how the composite design pattern provides this kind of functionality.



Example query:

“All products that are green and liquid:”

- Members of hierarchy level “products”
- Attribute filter “color = green”
- `IBICompositeMemberSetExpression` (join type = INTERSECT)
 - Members of hierarchy level “products”
 - Attribute filter “state of aggregation = liquid”

The example “all products that are green and liquid” shows three main steps:

1. Selecting all products of a specific level
2. Restricting this collection to only the ones that are green in color
3. Intersecting this restricted collection with another collection

But before you can intersect both collections, the collection represented by the `IBICompositeMemberSetExpression` must be computed first. For this purpose, you can associate the `IBICompositeMemberSetExpression (IBICompositeTupleSetExpression)` with a pair of parenthesis.

MemberSetExpressions

`IBIMemberSetExpressions` and related classes are diagrammed below:

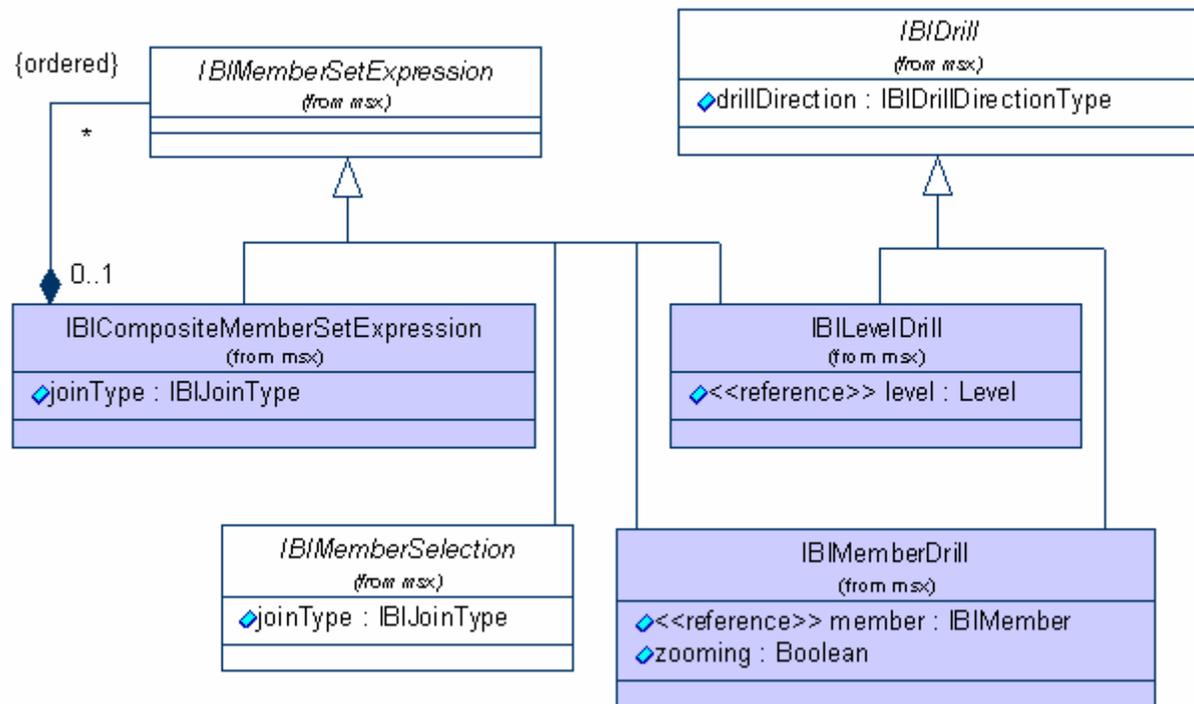


Figure 20 — MemberSetExpression Diagram

Note from the diagram that the four SetExpressions categories (see Categories of SetExpressions) are available:

1. Select:
IBICompositeMemberSetExpression, and all subinterfaces of IBIMemberSelection.
2. Filter:
IBIRankingFilter and IBIConditionBasedFilter
3. Sort:
IBISort
4. Drill:
IBILevelDrill and IBIMemberDrill

MemberSelection

MemberSelection is a MemberSetExpression. Below are the sub-classes of IBIMemberSelection, which are themselves MemberSetExpressions:

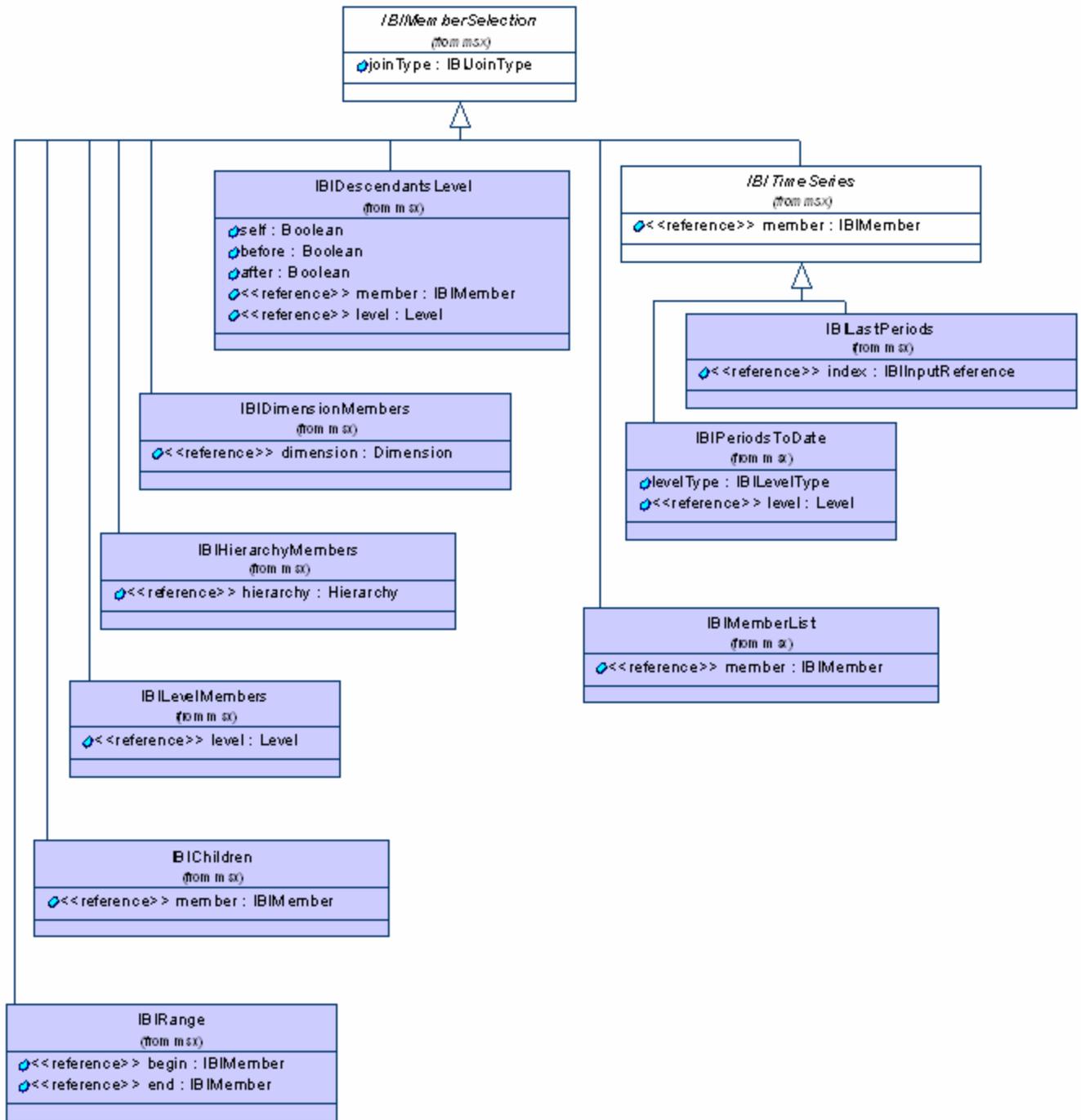


Figure 21 — MemberSelection Diagram



Note: Member selection in hierarchies

When considering member selection, it's important to consider the effect of the ALL member when creating queries. For a discussion, see Member Selection Based on Level, in the Accessing Metadata chapter.

TupleSetExpressions

Below, we diagram IBITupleSetExpression and its related classes:

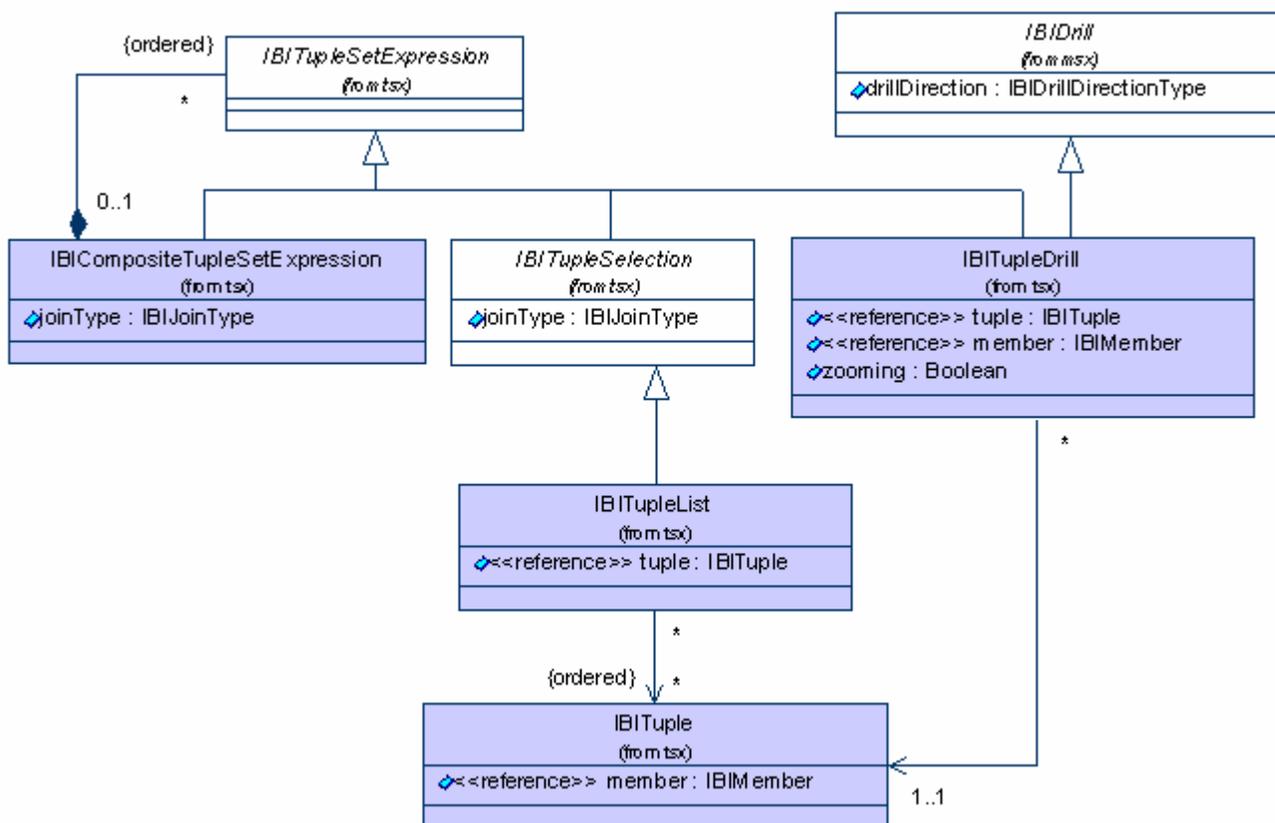


Figure 22 — TupleSetExpression Diagram

As with MemberSetExpression, the four categories of SetExpressions are available:

1. **Select:**
 IBCompositeTupleSetExpression, and all subinterfaces of IBITupleSelection (which is only IBITupleList).

Query APIs

2. Filter:
IBIRankingFilter and IBIConditionBasedFilter
3. Sort:
IBISort
4. Drill:
IBITupleDrill

Member

Many `SetExpressions` must be parameterized with `Members`. You do this with `IBIMember` or its sub-types. A member represents a single and unique value of a dimension. For example, a query has a customer dimension, which is the container for all existing customers of an enterprise system. One customer, such as “Kozmo USA,” would be represented by a single *member* of the customer dimension.

`IBIMember` and its associated classes are diagrammed below:

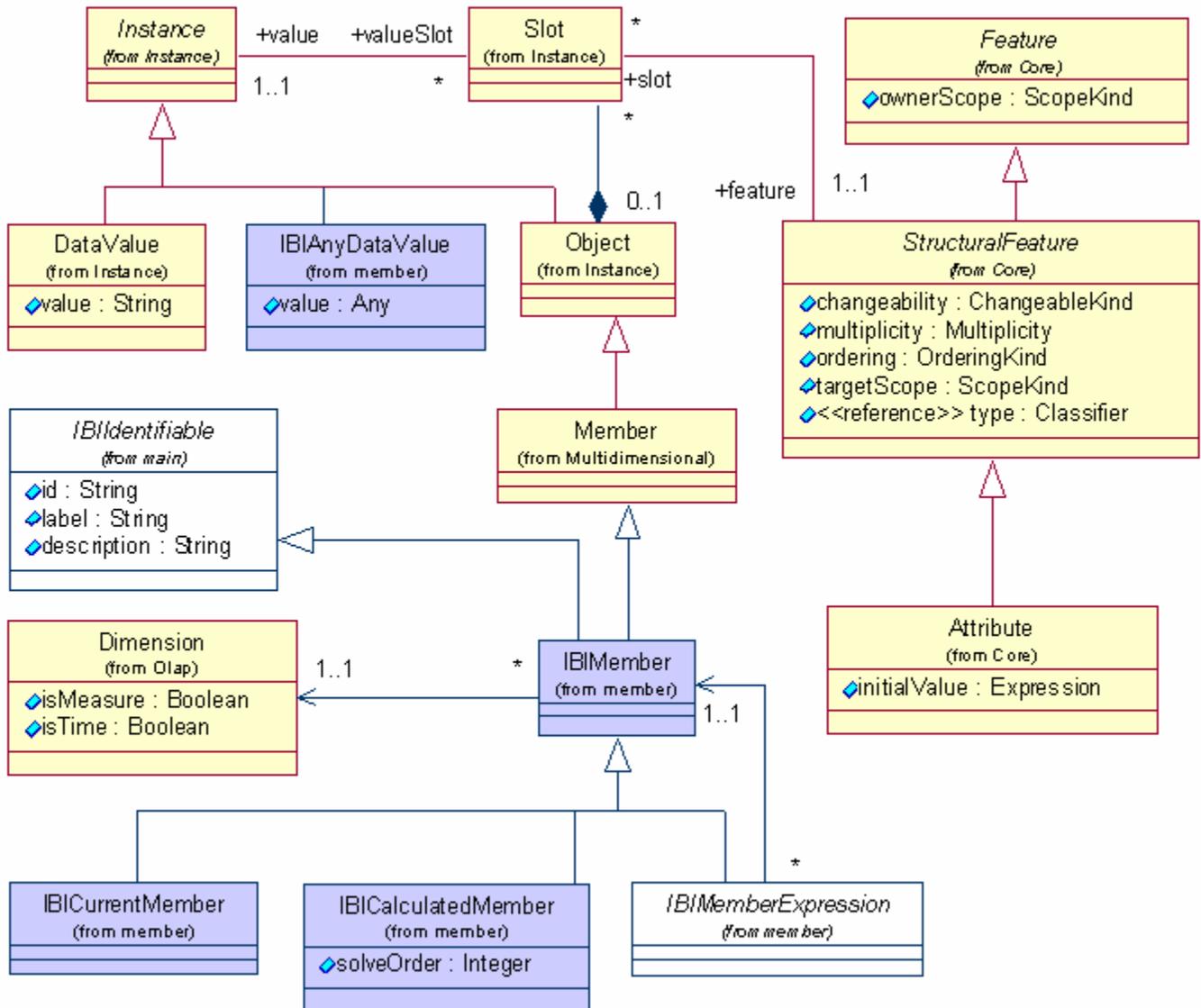


Figure 23 — Member Diagram

A customer – a member – is usually also associated with a collection of attributes such as ID, name, address, city, and zip code. These attributes are also accessible from the member object; from each member, you can request its associated attributes from the server, and these attributes will be returned with the result set.

In the OLAP Query API's com.sap.ip.bi.sdk.olap.query.member package, `IBIMember` is derived from the `Member` class defined in the CWM Multidimensional package. `IBIMember` (and its sub-classes) therefore derives all properties from `Member` (CWM).

Query APIs

A class called `Slot` models attributes and their corresponding values. `Object`, from which `Member` (CWM) is derived, aggregates a collection of `Slots`. A `Slot` can be regarded as an Attribute-Value pair. The `Slot` references an `Instance`, the value, and a `StructuralFeature`, which is the super class of `Attribute`.

There is only one data type defined in CWM that can be used for values in this scenario: the `DataValue` class, which is a subclass of `Instance`, and which has an attribute “value” of type `String`.

In order to support various data types for `Attribute` values, the `Member` package includes the `IBIAnyDataValue` class, which has an attribute of type `Any`. Any corresponds to the `Object` class in Java, which is used in the Java API to allow the following Java Wrapper classes as input:

- `Boolean`
- `Byte`
- `Character`
- `Double`
- `Float`
- `Integer`
- `Long`
- `Short`
- `String`

In the example of the customer Kozmo USA, the attributes of ID, name, address, city and zip code, would be represented by an instance of the `IBIMember` class, which aggregates a number of `Slots`. One `Slot` would have an association to an `Attribute` (for example, zip code) and an association to `IBIDataValue` (for example, value = 94114) for the value of the `Attribute`.

`IBIMember` also references the `Dimension` to which it belongs.

Several subclasses of `IBIMember` are special members:

- `IBICurrentMember`
This class represents the current member of a specific dimension while looping over a collection of members in the context of a query. Use when joining collections of members with `GENERATE` join type (see `Join Types`).
- `IBICalculatedMember`
This class represents a member that only exists within the context of a query (and not on the server). It is defined for a specific `Dimension`, and can have attributes like regular members. A calculated member is derived from existing values and is used for calculations such as sums, totals, subtotals, complex calculations based on other objects, and intermediate results (see also `InputReference`).
- `IBIMemberExpression`
(see below)

MemberExpressions

MemberExpressions represent single Members. They are used to dynamically refer to a member that has a relationship with another member that has to be specified. MemberExpression and its related classes are diagrammed below:

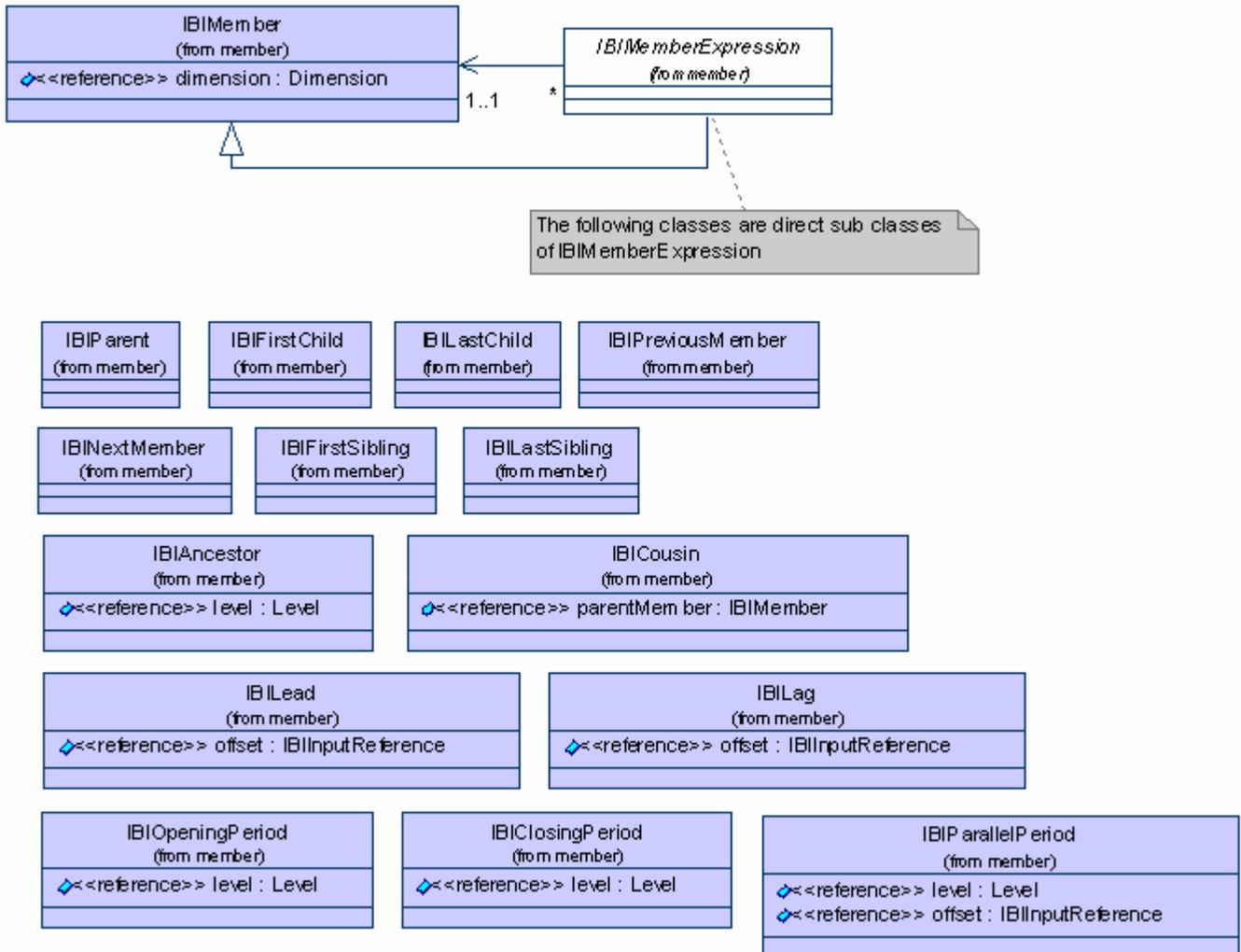


Figure 24 — MemberExpression Diagram

Consider, for example, the MemberExpression IBIParent in the diagram above. You must provide a member as an input to this expression. The MemberExpression IBIParent therefore represents the parent member of the member you have provided as input to this expression.

Query APIs

All the available `MemberExpressions` have in common that they represent a single member based on another member, and may be based additionally on levels and parameters of type `IBIInputReference`.

`IBIMemberExpression` is derived from `IBIMember`, and can therefore be used where ever a member is required.

InputReferences

All subclasses of type `IBIInputReference` can be used as input for another expression. They serve as inputs for other `IBIInputReferences`, `IBIMemberSelections`, `IBISetExpressions`, and `IBIMemberExpressions`, and as formulas for `IBICalculatedMembers`.

You specify a `CalculatedMember` by one input reference, which acts as its formula. There is only one formula per calculated member.

`IBIInputReference` and its related classes are diagrammed below:

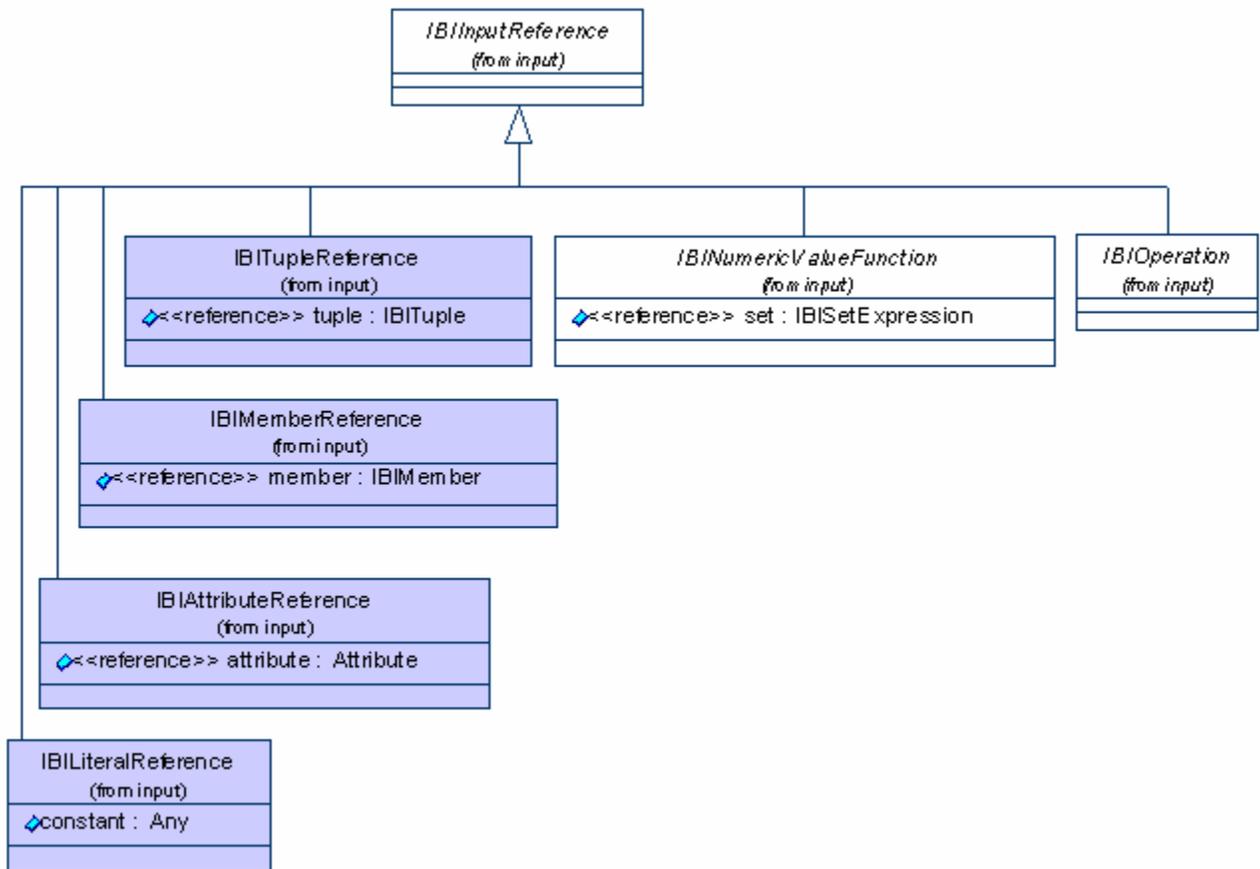


Figure 25 — InputReference Diagram

There are three kinds of InputReferences:

- Adapter classes
- NumericValueFunctions
- Operations

Adapter classes

Adapter classes, such as IBAttributeReference, are direct subclasses of IBInputReference and make the classes they adapt usable as input reference.

Operations

The third type, Operations, includes classes such as `IBIAddition`, which are the direct subclasses of `IBIOperation`, and represent unary and binary operations. In contrast to `IBINumericValueFunction`, `IBIOperation` does not work on a collection of tuples/member, but has one to two input parameters. It takes care of basic operations like addition, subtraction, multiplication, and division.

`IBIOperation` and its related classes are diagrammed below:

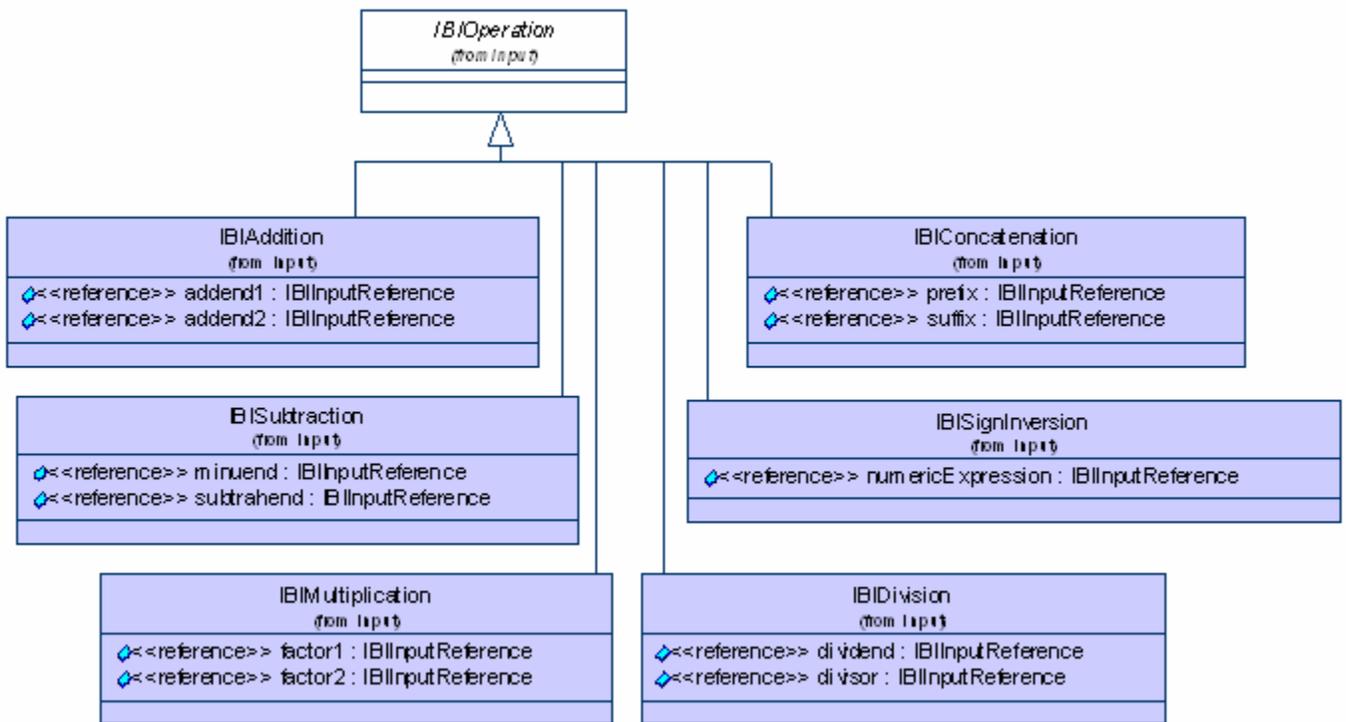


Figure 27 — Operation Diagram

Types

The OLAP Query Model defines several types, with their corresponding enumerations, that are used as type safe input for other expressions. They are diagrammed below:

Query APIs

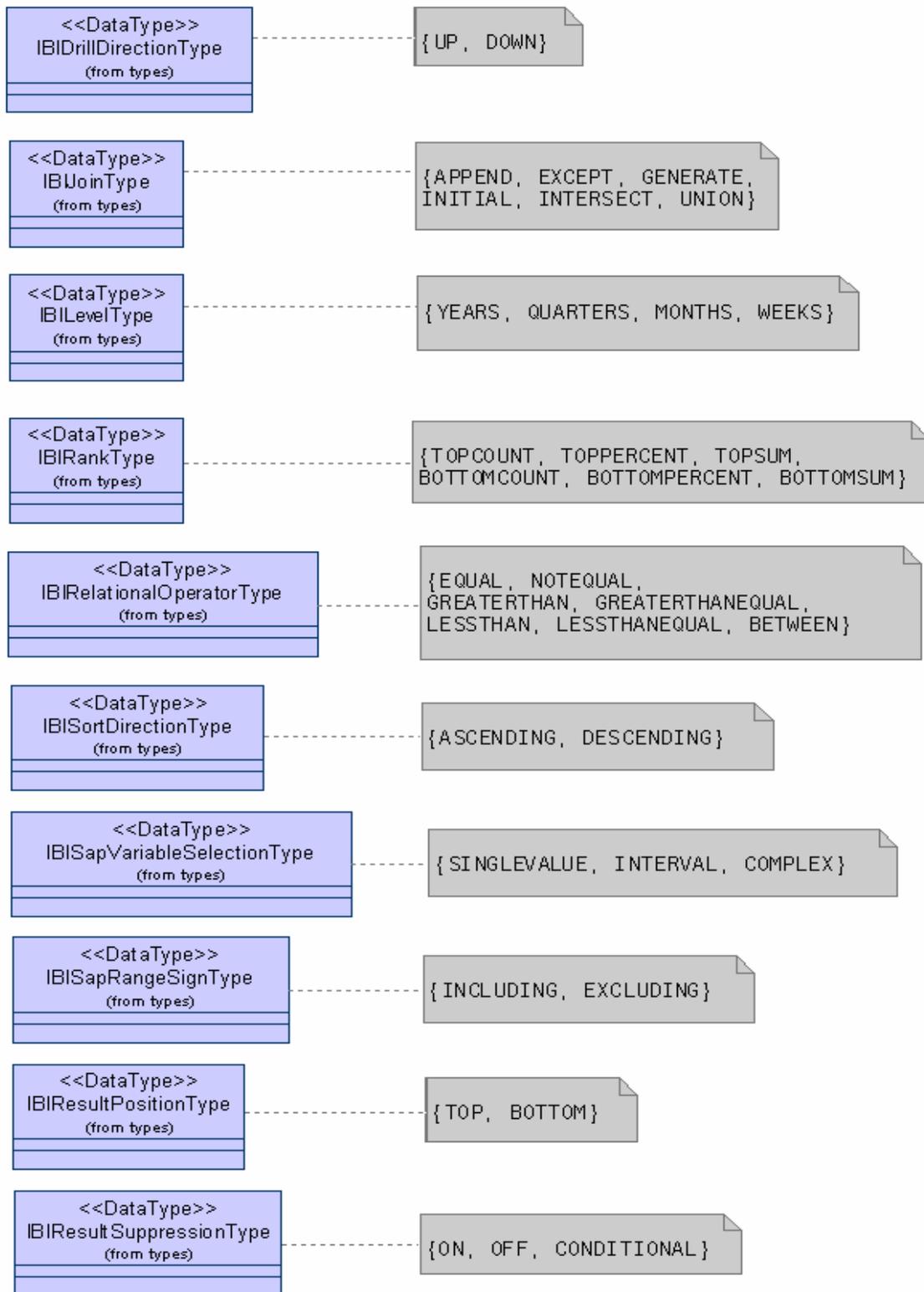


Figure 28 — Types Diagram



Note:

Two types, `IBIResultPositionType` and `IBIResultSuppressionType`, are not available in SAP NetWeaver 2004 or earlier versions.

Join Types referred to throughout the query model are also described here. For each join type, there is one enumeration value of the type `IBIJoinType`.

The available types and their enumerations are also listed in the table below:

Types and their enumerations

Type	Enumeration	Description
<code>IBIJoinType</code>	- APPEND - EXCEPT - GENERATE - INITIAL - INTERSECT - UNION	see Join Types.
<code>IBIRankType</code>	- TOPCOUNT - TOPPERCENT - TOPSUM - BOTTOMCOUNT - BOTTOMPERCENT - BOTTOMSUM	The type of ranking you can select for the ranking filter.
<code>IBIDrillDirectionType</code>	- UP - DOWN	The options for a drill operation.
<code>IBISortDirectionType</code>	- ASCENDING - DESCENDING	Specifies sort order.
<code>IBILevelType</code>	- YEARS - QUARTERS - MONTHS - WEEKS	Specifies a specific period of time.
<code>IBIRelationalOperatorType</code>	- EQUAL - NOTEQUAL	Specifies the relational operator for a condition-based filter.

Type	Enumeration	Description
	<ul style="list-style-type: none"> - GREATERTHAN - GREATERTHANEQUAL - LESSTHAN - LESSTHANEQUAL - BETWEEN 	
IBISapRangeSignType	<ul style="list-style-type: none"> - INCLUDING - EXCLUDING 	Specifies whether the given values should be included or excluded from the selection by using the SIGN component.
IBISapVariableSelectionType	<ul style="list-style-type: none"> - SINGLEVALUE - INTERVAL - COMPLEX 	Specifies one of three possible types of value selections for variables.
IBIResultPositionType	<ul style="list-style-type: none"> - TOP - BOTTOM 	Specifies the position in which to display the totals on an IBIAxis.
IBIResultSuppressionType	<ul style="list-style-type: none"> - SINGLEVALUE - INTERVAL - COMPLEX 	Specifies the conditions under which to calculate totals for an IBIAxisDimension.



Note:

Two types, IBIResultPositionType and IBIResultSuppressionType, are not available in SAP NetWeaver 2004 or earlier versions.

SAP Variables

In SAP BI, you can use variables to parameterize queries. For example, you could create a query for a monthly report, and then parameterize it with the current month using SAP variables.

Since these variables can be used in SAP BI's Open Analysis Interfaces, they therefore are supported in the BI Java SDK by using various BI-specific enhancements to the MDX syntax.

IBISapVariable and the additional classes related to SAP variable support in the OLAP Query Model are diagrammed below:

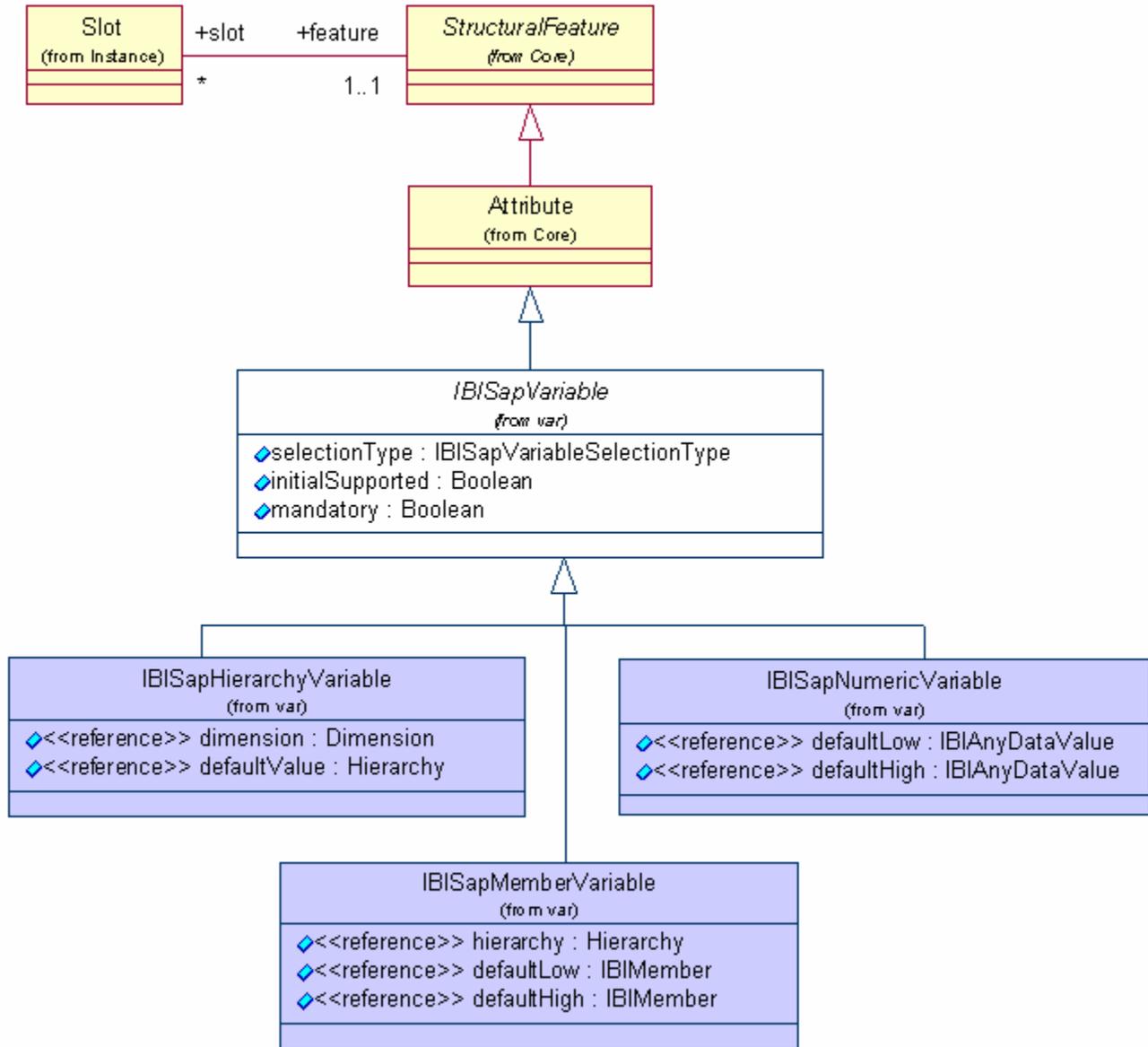


Figure 29 — Variables Diagram

An OLAP query in SAP BI (which corresponds to the CWM `Cube` in the query model) can have zero to many SAP variables associated with it. As you can see from the diagram, and as depicted in Figure 17 — Main Model Diagram above, support for this in the query model is provided by the `Slot` (see Member for more about slots). `IBIQuery` has a one-to-many association with `Slot`.

Each slot associates a variable with a value or a number of values. Variable values and their associated classes are diagrammed below:

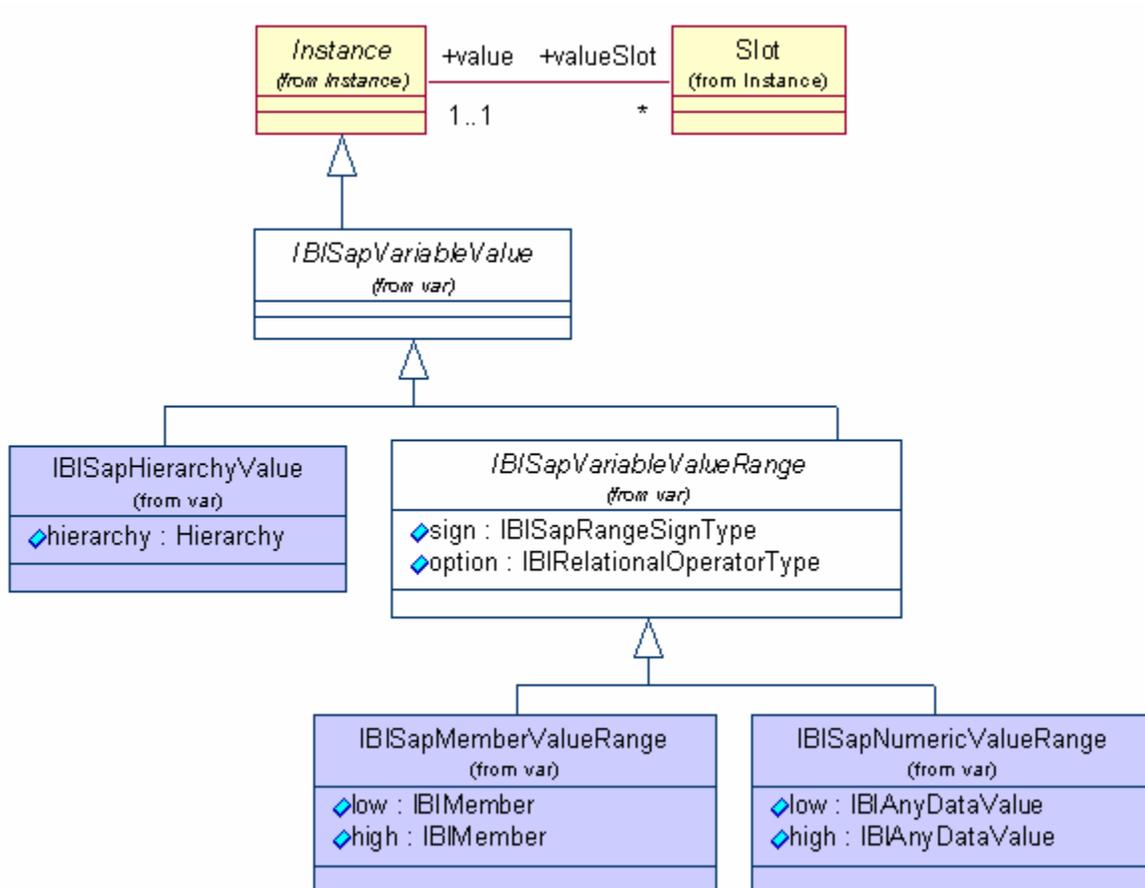


Figure 30 — Variable Values Diagram

Variable values are specified by selection types (see `IBISapVariableSelectionType` in the Variables diagram, above). There are three types of variables, and each specifies a particular selection type:

Variables and their selection types

SAP Variable	Selection Type
<code>IBISapHierarchyVariable</code>	Single value (you can specify only one hierarchy)
<code>IBISapMemberVariable</code>	Single value / interval / complex selection
<code>IBISapNumericVariable</code>	Single value / interval / complex selection

Single values simply select a single value, such as a `Member`. With an interval, you can select a range of values. With complex selection, you can select complex sets of collections of ranges and single values, including and excluding selections as well.

Relational Query Model

The Relational Query Model provides an abstraction layer for formulating relational queries independently of data source-specific query APIs. The model is based on the CWM Expressions package, and loosely on a subset of the SQL standard, and binds the CWM-based relational metadata.

A relational query in this sense is an expression tree of a functional representation of the corresponding SQL-like statement. This allows for simple code generation from the expression tree for the various relational backends (resource adapters, or BI Java Connectors), such as SQL for the BI JDBC Connector and an RFC call via JCo for the BI SAP Query Connector.

In contrast to the OLAP Query Model, the Relational Query Model is based almost purely on CWM, with some SDK extensions. It consists of the following CWM packages, and SDK extensions:

Relational Query Model: components and packages

Component	Associated CWM or SDK package
Foundation for the Relational Query Model	org.omg.cwm.foundation.expressions
Foundation for the Relational Metadata Model	org.omg.cwm.resource.relational
Foundation for CWM metamodel development	org.omg.cwm.objectmodel.core
Relational Command Processor SDK extensions to the CWM Expressions package	com.sap.ip.bi.sdk.dac.relational.query
Interfaces relevant for SAP Query	com.sap.ip.bi.sdk.dac.relational.query.sapq
Interfaces that help represent a WHERE condition in tree form	com.sap.ip.bi.sdk.dac.relational.query.tree

For API documentation, refer to the Javadocs for these packages via the links in the table above, or start with the Javadocs overview pages:

- [SDK Javadocs overview page](#): see Relational Query API group
- [CWM Javadocs overview page](#)

Relational Command Processor

The Relational Command Processor hides the complexity of the CWM Expressions package, which forms the basis of the Relational Query Model and assists in instantiating valid query instances. You can rely on the Relational Command Processor for most of your relational querying needs.

Consult the CWM Expressions package for additional underlying information about the Relational Query Model, or read additional background information below.

SQL Subset

The Relational Query Model corresponds with the following subset of SQL (specified as a grammar in *Backus Naur Form*):

Relational Query Model SQL subset (in *Backus-Naur Form*):

<code><single-row-select></code>	→	<code><select-clause> <from-clause> [<where-clause>] [<group-by-clause>] [<order-by-clause>] [<having-clause>]</code>
<code><select-clause></code>	→	<code>SELECT [<select-item>,*]<select-item></code>
<code><select-item></code>	→	<code><single-column> <all-columns></code>
<code><single-column></code>	→	<code><scalar-expr> AS <column></code>
<code><all-columns></code>	→	<code>[<range-var>.*]</code>
<code><scalar-expr></code>	→	<code><column-ref> <aggr-func-ref> <quoted-value> <numeric-value></code>
<code><aggr-func-ref></code>	→	<code><count-aggr-func-ref> <min-aggr-func-ref> <max-aggr-func-ref> <avg-aggr-func-ref></code>
<code><count-aggr-func-ref></code>	→	<code>COUNT(<column-ref>)</code>
<code><min-aggr-func-ref></code>	→	<code>MIN(<column-ref>)</code>
<code><max-aggr-func-ref></code>	→	<code>MAX(<column-ref>)</code>
<code><avg-aggr-func-ref></code>	→	<code>AVG(<column-ref>)</code>
<code><column-ref></code>	→	<code>[<column-quantifier>.]<column></code>
<code><where-clause></code>	→	<code>WHERE <cond-expr></code>
<code><group-by-clause></code>	→	<code>GROUP BY [<column-ref>,*]<column-ref></code>
<code><order-by-clause></code>	→	<code>ORDER BY [<order-item>,*]<order-item></code>
<code><order-item></code>	→	<code><column>[<order-dir>]</code>
<code><order-dir></code>	→	<code>ASCENDING DESCENDING</code>
<code><having-clause></code>	→	<code>HAVING <cond-expr></code>
<code><from-clause></code>	→	<code>FROM [<table-ref>,*]<table-ref></code>
<code><table-ref></code>	→	<code><table>[<range-var>]</code>
<code><catalog></code>	→	<code><catalog></code>
<code><schema></code>	→	<code>[<catalog>.]<schema></code>
<code><table></code>	→	<code>[<schema>.]<table></code>
<code><range-var></code>	→	<code><rangevar></code>
<code><column-quantifier></code>	→	<code><table> <range-var></code>

Query APIs

<i><column></i>	→	<column>
<i><cond-expr></i>	→	[<i><cond-term></i> OR] <i><cond-term></i>
<i><cond-term></i>	→	[<i><cond-term></i> AND] <i><cond-factor></i>
<i><cond-factor></i>	→	[NOT] <i><cond-test></i>
<i><cond-test></i>	→	<i><cond-primary></i> [IS NULL]
<i><cond-primary></i>	→	<i><simple-cond></i> (<i><cond-expr></i>)
<i><simple-cond></i>	→	<i><comparison-cond></i> <i><between-cond></i> <i><like-cond></i> <i><in-cond></i> <i><test-for-null-cond></i>
<i><comparison-cond></i>	→	<i><row-constructor></i> <i><comparison-operator></i> <i><row-constructor></i>
<i><comparison-operator></i>	→	<i><equals-operator></i> <i><less-than-operator></i> <i><greater-than-operator></i> <i><less-equals-operator></i> <i><greater-equals-operator></i>
<i><equals-operator></i>	→	=
<i><less-than-operator></i>	→	<
<i><greater-than-operator></i>	→	>
<i><less-equals-operator></i>	→	<=
<i><greater-equals-operator></i>	→	>=
<i><between-cond></i>	→	<i><row-constructor></i> BETWEEN <i><row-constructor></i> AND <i><row-constructor></i>
<i><like-cond></i>	→	<i><row-constructor></i> LIKE <i><row-constructor></i>
<i><escape></i>	→	ESCAPE <i><escape></i>
<i><in-cond></i>	→	<i><in-cond-list></i> <i><in-cond-sub-select></i>
<i><in-cond-list></i>	→	<i><row-constructor></i> IN ([<i><row-constructor></i> ,*] <i><row-constructor></i>)
<i><in-cond-sub-select></i>	→	<i><row-constructor></i> IN <i><single-row-select></i>
<i><test-for-null-cond></i>	→	<i><row-constructor></i> IS NULL
<i><row-constructor></i>	→	<i><scalar-expr></i>
<i><quoted-value></i>	→	<quoted>
<i><numeric-value></i>	→	<numeric>

Notes:

1. Non-terminals are in italics, for example: *<non-terminal-symbol>*.
2. Terminals are in bold, for example: **<terminal-symbol>**.
3. Keywords are in bold, for example: **KEYWORD**.
4. "|" separates alternative grammar productions.
5. "[...]" encloses optional grammar productions.
6. "*" specifies that the preceding grammar production can occur an arbitrary number of times.
7. The terminal symbols have the following meaning:

Terminal Symbols

Terminal Symbol	Corresponding Java Type
<catalog>	String (valid catalog name, if supported)
<schema>	String (valid schema name, if supported)
<table>	String (valid table name)
<rangevar>	String (valid range-var string)
<column>	String (valid column name)
<escape>	char
<quoted>	String surrounded by single quotes
<numeric>	int, double

8. The semantics of a query instance correspond with their SQL counterpart.
9. While WHERE conditions are tree-like expressions (Backus-Naur Form, referenced above, implies binary logical operators and enclosing parentheses), the Relational Command Processor implies a sequential model of query generation and manipulation, hence the WHERE is formulated using a logically equivalent postfix representation (a stack-based approach). In other words, the predicates are pushed onto an implied WHERE stack followed by the logical operators. Refer to the Javadoc for the Relational Command Processor for more information.

SAP Query Interfaces

SAP Query is a reporting tool for systems running SAP Web Application Server, and is also accessible via an ABAP BAPI. This package provides interfaces to the data structures used when accessing SAP Query via this BAPI (Functiongroup RSAQ).

Metadata Mapping

SAP Query uses the following metadata objects:

- Functional area
 - A functional area describes a data source (field list and methods for data access). The access methods are can be SQL statements, logical databases and ABAP reports and are hidden for users of the functional area.
- Query
 - A query uses a functional area to select the fields which are to read from the data source described by functional area. Each query must use exactly one functional area; a

Query APIs

combination of functional areas like joins is not possible. Additionally, a query is connected with a user group.

- User group

A user group is a container for queries and an authorization mechanism for using SAP Query. User groups are not used in the BAPI and as support for them will be dropped, their use is not recommended.

These objects have been mapped to the relational model used in the SDK in the following manner:

SAP query / Relational Model Mapping

SAP Query	SDK Relational Model	Description
SAPQUERY	Catalog	<i>placeholder catalog</i>
SAPQUERY	Schema	<i>placeholder schema</i>
Functional Area	Table	
Functional Area Fields	Column	

Usergroups could have been a potential candidate for either catalog or schema, but as their support will be discontinued, this was not chosen.

As the results of generic queries in SAP Query can be far more complex than the mapping to a relational table would allow (for example, several separate results as the result of one query) this was not feasible within the framework of the relational query model of the SDK. Therefore, Functional Area was chosen as the representation of a relational table instead, still allowing full access to the entities offered within SAP Query, with the restriction however that queries need to be generated for a Functional Area via the SDK.

SAP Query does not support queries involving more than one functional area at once (in other words, JOINS are not possible).

SELECT-OPTIONS

As SAP Query uses the runtime of the underlying SAP Web Application Server and provides access not only to relational tables, but also to Logical Databases and Reports via the same BAPI, query selections and restrictions (specified as WHERE conditions in SQL) are limited to a subset of the expressions permissible in the relational model. Where possible, WHERE conditions are translated to their equivalent representation in SAP Query called SELECT-OPTIONS, or else an exception is raised. Details of the subset are can be found in the SAP Query sub-package API documentation of the Relational Query API package group, as well as in the SAP online help for the SAP Web Application Server at the following link:

http://help.sap.com/saphelp_webas620/helpdata/en/9f/dba71f35c111d1829f0000e829fbfe/frameset.htm

Query Execution

SAP Query query execution supported via the SDK consists of the following phases:

1. SAP Query query based on the functional area and metadata provided (fieldlist for the selection, etc.) is generated on the fly. SAP Query query is executed with the SELECT-OPTIONS restrictions (converted from the original relational form of the relational query).
2. SAP Query result is retrieved (simple text-based JCo table) and converted to a relational result set (SDK implementation of java.sql.ResultSet) and returned. Due to the implementation of the BAPI, the entire result of such a query is retrieved at once (even though java.sql.ResultSet would allow a cursor-based fetch model).

Tree-Form Representation

With the Relational Query API, logical expressions used to specify WHERE conditions can be represented as trees containing the logical operations AND, OR, and NOT as internal nodes and predicates, such as a=1, as the leaves.

The predicates usually are of the form:

<column> <oper> <value>

In the above case, <value> can be one of the following:

- a literal value containing digits or a string (for example: 1, '01-01-2004', 'text')
- another column (to create joins)
- a sub-select



For example:

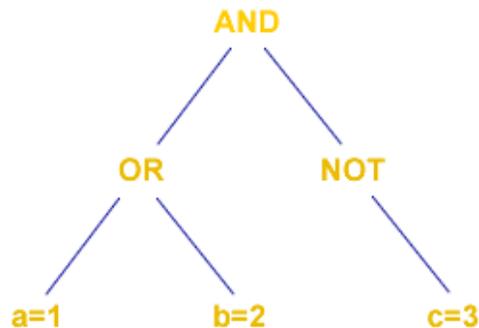


Figure 31 — Tree Form Representation

Examples

The where condition specified by the above tree is:

```
( (a=1) OR (b=2) ) AND (NOT (c=3) )
```

This tree could be implemented using the following code:

```
Column a = ...;
Column b = ...;
Column c = ...;
BIWhereTree tree = new BIWhereTree();
tree.push(a, IBISQLComparisonOperator.EQUALS), new Integer(1));
tree.push(b, IBISQLComparisonOperator.EQUALS), new Integer(2));
tree.push(IBISQLLogicalOperator.OR);
tree.push(c, IBISQLComparisonOperator.EQUALS), new Integer(3));
tree.push(IBISQLLogicalOperator.NOT);
tree.push(IBISQLLogicalOperator.AND);
tree.popRoot();
```

Examples

Although most of the SDK's examples create queries in some way, the examples below focus in particular on query functionality.

OLAP Queries



Olap_2.java – Direct execution of MDX statement:

Demonstrates how to retrieve a result set by directly executing an MDX statement, then shows how to display the result set as an HTML table.



Olap_3.java – Pivoting / changing layout of an OLAP query:

Illustrates the process of changing the layout of a query by moving dimensions between axes and then by swapping axes. Renders the output of each into two separate HTML tables.

To help visualize the effects of pivoting dimensions on axes, we illustrate the process in a series of four diagrams below:

Examples



Figure 32 — Operation: New Query

MDX: SELECT FROM Sales Cube

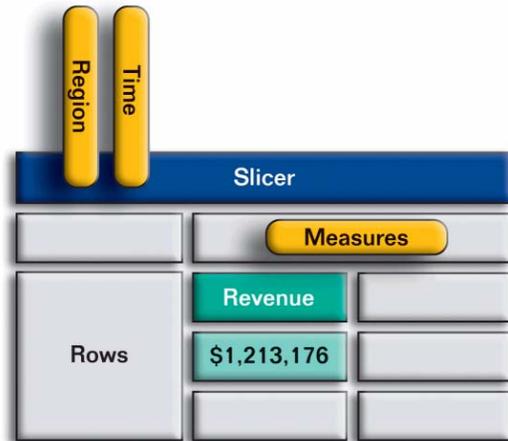


Figure 33 — Operation: moveDimensionToColumns

MDX: SELECT { [Measures].[Revenue] } ON COLUMNS FROM SalesCube

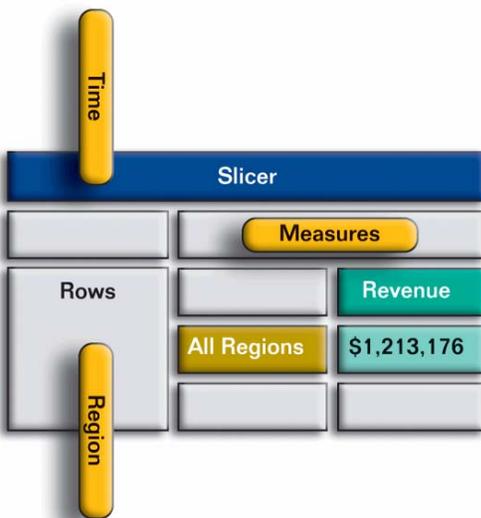


Figure 34 — Operation: moveDimensionToRows

MDX: SELECT { [Measures].[Revenue] } ON COLUMNS, { [Regions].[All] } FROM SalesCube

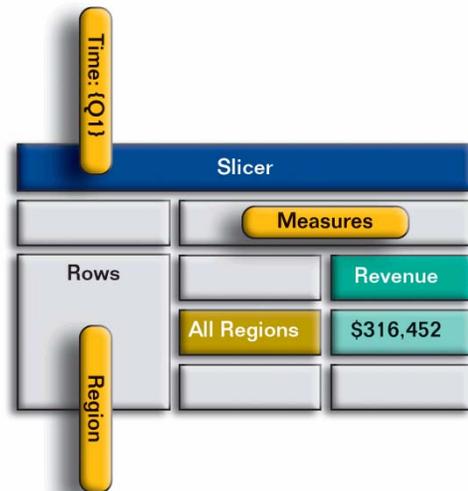


Figure 35 — Operation: addMember

MDX: SELECT { [Measures].[Revenue] } ON COLUMNS, { [Regions].[All] } FROM SalesCube WHERE ([Time].[Q1])

As diagrammed in the series above, the result of a query in its initial state (see Figure 32 — Operation: New Query, above) orients all dimensions on the slicer axis with their respective default members of default hierarchies selected.

Examples

In this state, the result of the query selects a single cell from the cube. The MDX statement, more specifically, is equivalent to:

```
SELECT FROM Sales Cube WHERE ([Region].[All],[Time].[All],[Measures].[Revenue]).
```

To display a certain dimension in the report, we must move it from the slicer axis to one of the other axes (columns or rows) (see Figure 33 — Operation: `moveDimensionToColumns`, above). The result of this operation moves the measures dimension to the columns axis and selects its default member, which is `Revenue`.

Next, we move a different measures dimension to rows (see Figure 34 — Operation: `moveDimensionToRows`, above). Now the `Region` dimension is moved to the rows axis and selects the default member. The data set in this state is a typical crosstab, however it displays only a single member on each axis.

Finally, to finish refining our result, we use the `addMember` operation (see Figure 35 — Operation: `addMember`, above). In this figure, a member is added to the member set of the `Time` dimension on the slicer axis. This results in the specification of a filter value in the `WHERE` clause, and rounds out a specific result set with which we are concerned.



Olap_4.java – Selecting dimension attributes:

Selects a dimension attribute, and renders the result set into an HTML table.



Olap_5.java – Sorting by measure value:

Renders the default result set into an HTML table, sorts the data according to a measure value in ascending order, and then renders the data into a second HTML table for comparison.



Olap_6.java – Sorting by dimension attribute:

Illustrates how to select a dimension attribute for display, and to sort by a dimension attribute.



Olap_7.java – Filtering:

Illustrates both a ranking filter and a condition-based filter. Renders the result of a query without any filtering, then filters the set of `Sold-To` parties using a ranking filter and re-renders the result. Changes the filter to a condition-based filter to restrict by quantity, and re-renders the result for comparison.



Olap_8.java – Hierarchy navigation – member drill operations:

This example illustrates the two different versions of hierarchy navigation: zooming and drilling.

Examples

When zooming in on a member, the member itself is discarded and only the children are displayed. The advantage of this method is that the result sets remain simple. The disadvantage is that the context of where the members belong in the hierarchy is lost. When drilling down on a member, the member itself is retained and its children are added to the result set.

In this example, we apply the following operations in sequence to an initial data set:

1. Zoom in
2. Zoom out
3. Drill down
4. Drill up

After each operation, the result set is rendered again for comparison.



Olap_9.java – Calculated members:

Creates a calculated measure - cost per item - by dividing the total cost by the number of items sold.



Olap_10.java – SAP variable complex value selection:

Illustrates two ways to define a complex value selection for an SAP variable:

1. Using the OLAP Command Processor
2. Directly manipulating the OLAP Query Model

Relational Queries



Relational_3.java – Direct execution of SQL statement:

Demonstrates how to retrieve a result set by directly executing a SQL statement, then shows how to display the result set as an HTML table.



Relational_4.java – Simple relational query:

Demonstrates how to retrieve a result set by creating a simple query, then shows how to display the result set as an HTML table.



Relational_5.java – More complex relational query:

Examples

This example in particular demonstrates the benefit of the SDK's query APIs and how you can build a lengthy, complicated SQL query without having to key it all in yourself and validate its syntax.

 **Caution:**

This example finds all tables in your database starting with a particular prefix. This could retrieve a great deal of data and cause performance issues if there are a large number of tables in your database and the prefix is not restrictive enough.

Here, we retrieve a result set by creating a more complex query with the following features:

1. Field selections
2. Joins
3. Sorting

 **Note:**

See Appendix B: Examples for the full index of examples and instructions on getting your system up and running with them.

Chapter 5: Retrieving Result Sets

Overview

In the analytical applications you create with the SDK, the ultimate goal of connecting to a data source, browsing its metadata, and creating a query upon it is to render a result, or result set. To create this result set, and to assist you in navigating and rendering it, the SDK provides the ResultSet API and the OLAP Table Model. This chapter describes the following components:

- ResultSet API
 - Key Features
 - OLAP Result Sets
 - Relational Result Sets
- OLAP Table Model

API Documentation

Refer to the Javadocs for the ResultSet API in the following package of the SDK:

com.sap.ip.bi.sdk.dac.result

Refer to the Javadocs for the OLAP Table Model in the following package of the SDK:

com.sap.ip.bi.sdk.dac.result.model

ResultSet API

The BI Java SDK needs to be able to represent result sets for the diverse queries that can be defined using the SDK's Query APIs. These result sets can have any geometry (for example, crosstabs with nested dimensions) and can contain any data type (for example, Time, Date, Timestamp, Double, String, and Integer), and can be relational or OLAP result sets.

It's the job of the SDK's Resultset API to provide applications with a complete set of interfaces to access these result sets, delivering a relational result set from a relational data source, and an OLAP result set from an OLAP data source.

ResultSet API

The ResultSet API is based on Sun's established `java.sql.ResultSet` interface, which is part of the Java Database Connectivity (JDBC) API. Only a few lightweight interfaces have been added to facilitate a multidimensional representation.

Key Features

Key features of the ResultSet API include:

- **Relational support**
The ResultSet API's basis in JDBC is grounded in supporting the representation of relational query results and is by nature tabular, supporting access to columns and navigation on the rows.
- **Multidimensional support**
The ResultSet API has additional elements that facilitate a multidimensional presentation. Although any result set can be represented as a flat table, it's too great a burden on OLAP applications to decompose such a representation to create a multidimensional rendering. To support multidimensional results, we have therefore added elements such as axes that correspond to the elements of a multidimensional query.
- **Performance**
The result space represented by a query can be very large. It can be impossible for an application to deliver so much data to a UI all at once. To avoid this, the ResultSet API allows an application to fetch only a part of the data that is represented by a query.
- **Hierarchies**
The API facilitates the rendition of hierarchies by providing information about hierarchy levels such as parent/child relations and number of children.
- **Cells**
The API supports provider-specific cell properties, such as currency/unit and numeric formatting information. The API supports empty cells and cells in which errors occurred, such as overflows.
- **Compatibility**
The API has been engineered to be compatible with a variety of data sources, supported by the SDK's ODBO, XMLA, JDBC, and SAP Query connectors. Since `java.sql.ResultSet` is not the native format for retrieving results from data sources that are not JDBC-compliant, support in the non-JDBC connectors for JDBC's features depends on the data source. For example, JDBC-specific data types such as `BLOB` are not supported.



Note:

We have inherited some functionality from `java.sql.ResultSet` that is not implemented in this version of the SDK:

- The `java.sql.ResultSet` API provides update, insert, and delete functionality which is not supported in the current implementation of the SDK.
- The API provides an association with the query. In other words, the API provides a means of navigating back to the query from which the ResultSet was created.

OLAP Result Sets

A data set is a multidimensional result set returned by an OLAP server. It is a complex data object that can be conceptually represented by two components: cell data and axis data. The `IBIDataset` interface provides the methods to access the information contained in a data set, supporting the concept of *cursors* (instances of `java.sql.ResultSet`) for each axis and dimension.

The data set representation conforms closely to conventions in Microsoft's OLE DB for OLAP (ODBO) Programmer's Reference. We've made adjustments in structures and naming to comply with Java naming conventions as well as for simplification.

The root interface for accessing a data set is the `IBIDataset` interface. A reference to `IBIDataset` is returned either by calling the `getDataset()` method on a query object that has been executed, or by executing an MDX command directly, using the method on an instance of the `IBIConnection` interface. `IBIDataset` provides access to the list of axis-cursors. These in turn provide access to `DimensionCursors` which can contain `DimensionAttributeCursors`.

All cursors implement `IBIResultSet` (extending `java.sql.ResultSet`), thus we have a very symmetric and simple design. The `DimensionCursor` provides a mechanism for separating axis data for the different dimensions and does not support row navigation. Instead, a `DimensionCursor` governs only a single row that corresponds to the current position of the `AxisCursor`.

In addition, we defined a mechanism to support basic manipulations of a data set, such as update of cell values, insertion and deletion of rows, and features like conditional sum or zero suppression. The concept of pluggable filters provides a good separation of such functions from the basic infrastructure of representing a general dataset. We've also retained the `insert`, `update`, and `delete` methods of `java.sql.ResultSet`, for basic manipulations of an in-memory-instantiated data set.

Finally, similar to ActiveX Data Objects (ADO) design, we have added a properties collection to the base `IBIResultSet` interface that will allow us the flexibility to add information about the quality of an instance of a `ResultSet`.

`IBIDataset` extends `java.sql.ResultSet` to provide direct access to the cell data (cell cursor) as well as to provide methods to retrieve the list of axes (axis cursors). `IBIAxisCursor` extends `IBIResultSet`, hence providing access to and navigation on the contents of each axis. The `AxisCursors` in turn contain `DimensionCursors` (one for each dimension) which in turn can contain `DimensionAttributeCursors` (for attributes of a dimension). For example, the dimension `Product` might have attributes `Color` and `Size`.

This relationship between axis, dimension, and cell cursors is schematically depicted below:

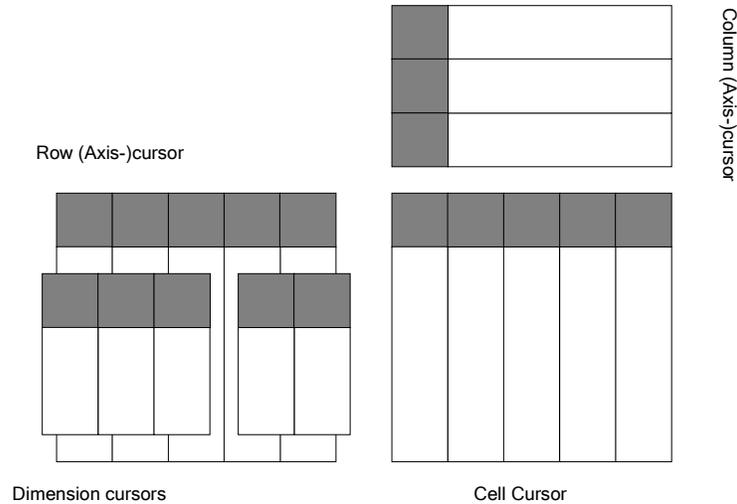


Figure 36 — ResultSet API Cursor Model

Major Interfaces Summary

The `IBIDataSet` interface provides the following features:

- Extends `IBIResultSet` to allow access to the cell data (referred to as a cell set).
- Provides that the cell set is access-only result set (as navigation is prompted by navigation of the axis cursors).
- Provides access to all axes.
- Provides synchronization between cell set and all the axes. When navigating along an axis, the cell set is positioned accordingly.

The `IBIAxisCursor` interface provides the following features:

- Extends `IBIResultSet` to allow navigation along each individual axis (navigation-only resultset).
- Contains dimensions (naming convention of columns to discern between dimensions, or dimension-specific prefix – also known as dimension cursors).

ResultSet API

- Provides synchronization between cell set and axis cursor.

The `IBIDimensionCursor` interface provides the following features:

- Extends `IBIResultSet` to provide access to dimension data (access-only result set).
- Provides for the access-only result set (navigation is handled via enclosing `AxisCursor`).
- Contains (optional) dimension attributes (naming convention to discern between attributes, or attribute specific prefix – also known as dimension attribute cursors).

The `IBIDimensionAttributeCursor` interface provides the following features:

- Extends `IBIResultSet` to provide access to dimension attribute data.
- Provides for the access-only result-set (navigation is handled via enclosing `AxisCursor`).

Metadata Access

Since `IBIDataSet` extends `IBIResultSet`, which in turn extends `java.sql.ResultSet`, it provides the same mechanism to access metadata describing the result set. `Metadata (java.sql.ResultSetMetadata)` is retrieved via the `getMetadata()` method, which provides the following information:

- Catalog, schema, and table name (applicable in the relational domain where possible – no table name is available if the result set is the result of a JOIN query)
- Number of columns in the result set
- Captions and names of the columns
- Column type information (data type and precision, length, and so on, where applicable)
- Additional features of the column (for example, can the column contain NULL values, is it writeable, and so on)

Metadata is provided for and needs to be retrieved separately from each component of the `IBIDataSet` instance – in other words, from the cell set, axis cursors, dimension cursors and dimension attribute cursors – via the same method on each cursor.

Navigational Aspects

As mentioned above, `IBIDataSet` supports navigation along each of the axis cursors, with automatic synchronization of the cell cursor, which eliminates the need to calculate the current position in the cell set based on the current positions in each of the axis cursors. To ensure consistency, the cell cursor does not allow the use of any

ResultSet API

of the `java.sql.ResultSet` navigation methods (attempts to use these would result in a `java.sql.SQLException`).

`IBIDataset` provides (via `java.sql.ResultSet`) a simple and effective navigation model (a cursor-based approach) supporting blockwise data fetching, including:

- Relative and absolute positioning via row index
- Simple stepping to next and previous row
- Defined reset of the navigation state (position after last, before first, on first and on last row)
- Fetchsize can be recommended (`set-/getFetchsize` method)
- Automatic synchronization of cell and axis cursors on change of navigation state

Data Access

`IBIDataSet` provides (via `java.sql.ResultSet`) typed access to the contents of the columns of the current row, provided the current row is valid (in other words, not before first or after last row). The column requested can be specified either by index (1-based – the first column has index 1, the second has index 2, and so on), or by column name.

Information on number and names of the columns is contained in the metadata of the result set. The data contained in each column can be retrieved by specifying the return type – `getString(int)` (column specified by index) and `getString(String)` (column specified by name).

`IBIDataSet` supports (via `java.sql.ResultSet`) access for all primitive data types, such as `boolean`, `int`, and `char`, as well as for a large collection of commonly-used classes, such as `String`, `Double`, `Float`, `Date`, and `Time`. The implementation of `java.sql.ResultSet` is responsible for converting the underlying data provided by the data source to the requested type and returning an appropriate value – in other words, for mapping the data type in the data source to a corresponding Java data type.

`IBIAxisCursor`, `IBIDimensionCursor` and `IBIDimensionAttributeCursor`, like `IBIDataset`, are extensions of `java.sql.ResultSet` and all provide the same interface for metadata access, navigation and data access.

Multidimensional Results in Two Dimensions

There are two basic approaches to representing multidimensional datasets in tables of two dimensions: data set flattening, and data set decomposition. The `ResultSet` API uses the data set decomposition approach. We compare the two and explain our motivation below.

Data Set Flattening

Flattening represents any result set as a flat table, and the process that renders such a representation of an OLAP data set is known as data set flattening. The ODBO flattening algorithm defines the construction of column names and also the rendition of hierarchy information in such a flattened data set.

The screenshot below shows an example of a flattened data set as returned by an ODBO provider:

[Gender].[Gender].[MEMBER_CAPTION]	[Marital Status].[Marital Status].[MEMBER_CAPTION]	[Measures].[Unit Sales]	[Measures].[Sales Average]
F	M	65336	6.53961945031712
F	S	66222	6.54554952195306
M	M	66460	6.49385701676962
M	S	68755	6.46019217936742
	M	131796	6.51660889470241
	S	134977	6.5019663827371

Figure 37 — Flattened Data Set

The creation of such a flattened data set comes with significant overhead for the provider. It also puts the burden of decomposing this representation on the OLAP application. The rationale for this approach is rooted in the legacy of some BI tools which initially were pure relational reporting tools, and then later added OLAP support. For such tools, a tabular representation was the easiest way to load data into their native data stores.

Data Set Decomposition

The SDK instead uses data set decomposition as its approach to rendering multidimensional result sets. Decomposition is the basic underlying concept of the ODBO `IMDDataset` interface and the ADO MD cellset object. In this approach, we decompose a data set into cell data and axis data, which we in turn represent by tabular views.

The figure below shows a sample dataset that consists of two axes – a columns axis and a rows axis:

	Store Cost	Store Sales
Berlin	€63,530.43	€159,167.84
Hamburg	€56,772.50	€142,277.07
Munich	€105,324.31	€263,793.22

Figure 38 — Sample Data Set

ResultSet API

Note that the concept allows for more than two axes, however a two-dimensional, table-like data set makes the example easy to illustrate. On the columns axis, two members ("Store Cost" and "Store Sales") of the measures dimension have been selected; on the rows axis, three members ("Berlin," "Hamburg," and "Munich") of the City level of a geographical hierarchy. The dataset has six cells:

Cells provide four mandatory properties:

- **Value** — supports all common column types, for example:
 - numeric types
 - dates
 - time values
 - strings
 - null
- **Data type** — int value describing the data-type (see `java.sql.Types`)
- **Status** — state of the cell (for example, error or null)
- **Formatted value** — a string representation of value

For example, the cell with the ordinal 1 has the value "159167.84" and the formatted value "€159,167.84." Additional properties of a cell are modeled as a properties collection of a cell.

The following figure now describes this information within the context of the decomposition of the data set into axis data, cell data, and collections:

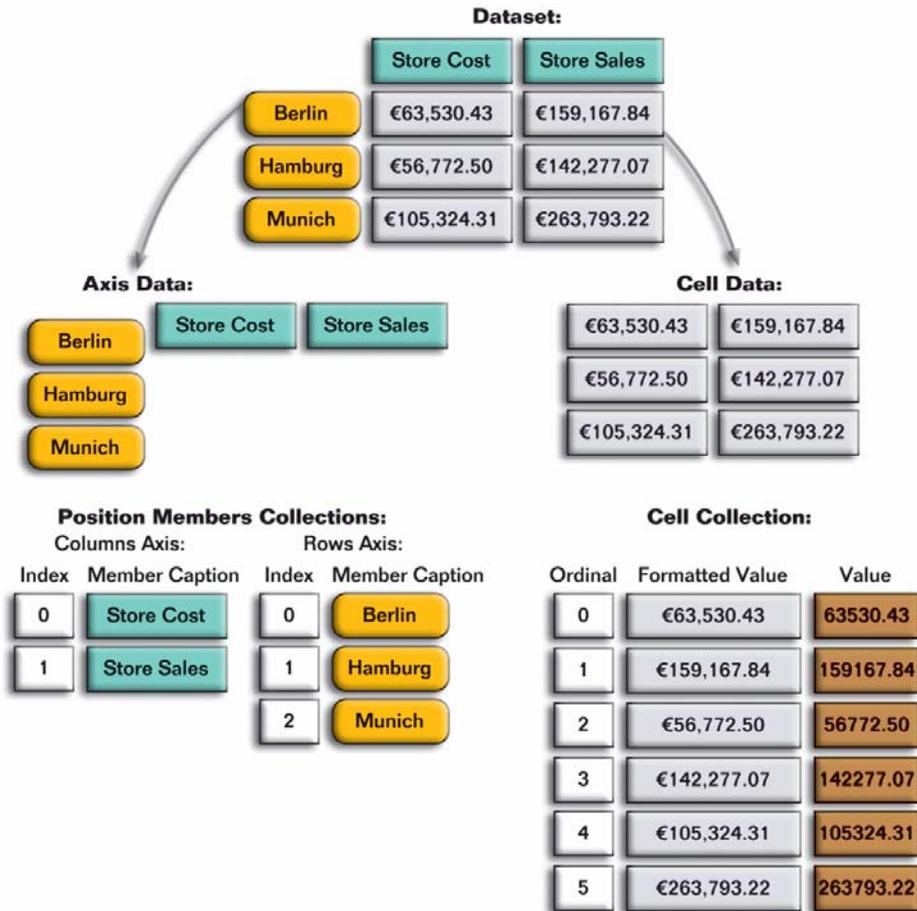


Figure 39 — Decomposed Data Set

The data set has been decomposed into three tabular views or collections:

1. Cell data
2. Members on the columns axis
3. Members projected along the rows axis

The example is fairly simple, insofar as there are no nested dimensions or attributes on either of the axes, and no additional properties for the cell data beyond the value and a formatted value. However, the extension of this model to more complex result sets with such properties is straightforward.

Relational Result Sets

The relational result set (`IBIResultSet`) is based on `java.sql.ResultSet`, from Sun's established standard. It supports all common data types and blockwise fetching of tabular data within a Java API.

Part of the set JDBC APIs, `java.sql.ResultSet`'s implementation is the responsibility of the JDBC driver implementers. We provide our own implementation for such cases where no JDBC driver is available, for example in the case of the BI OLAP and SAP Query connectors.

The `ResultSet` API's `IBIResultSet` extends `java.sql.ResultSet`, adding the capability to synchronize the navigation of multiple result sets for use in OLAP result sets. We also support the need to provide additional features, such as conditional zero suppression or summation and calculated keyfigures (additional columns based on data contained in the result set itself). To achieve this, `IBIResultSet` supports pluggable filters which follow the "decorator" design pattern, forwarding `IBIResultSet` calls to another `IBIResultSet` and adding the required functionality.

By reusing the `ResultSet` interface and all data types of the `java.sql` package, our design required adding only a few lightweight interfaces. In this way, we gain interoperability with the existing ecosystem for `java.sql.ResultSet`.

Metadata Access

As `IBIResultSet` extends `java.sql.ResultSet`, the standard means for accessing metadata of a result set is identical in both cases. The `getMetadata` method returns an instance of `ResultSetMetadata`, which provides information about:

- Catalog, schema, and table name (where applicable – and table name is "" when the result set is the result of a JOIN of multiple tables)
- Number of columns in the result set
- Caption and name of each column
- Additional features of the column (for example, can the column contain NULL values, is it writeable, and so on)

Navigational Aspects

As above, `IBIResultSet` (via `java.sql.ResultSet`) supports cursor-based navigation with

- Absolute and relative positioning via row index (1-based – the first row has index 1, the second has index 2, and so on)
- Single stepping forwards and backwards (next and previous row)
- Setting of specific positional states (before first, on first, on last, after last row)
- Querying of the current position (is the current row before first, after last, on the first, on the last row, and so on)

Data Access

As above, `IBIResultSet` (via `java.sql.ResultSet`) provides access to the contents of the current row (if it is valid, or not positioned before the first or after the last row). The contents of a column of the current row can be retrieved either via its index (1-based) or its column name.

`IBIResultSet` provides typed access to the data for all primitive types, such as `boolean`, `int`, and `char`, as well as to most common classes such as `String`, `Date`, and `Time`.

The mapping of data source types to Java language types is described in the JDBC specification (see <http://java.sun.com/products/jdbc/>).

OLAP Table Model

The OLAP Table Model is a helper class that facilitates the rendering of a multidimensional dataset into a two-dimensional matrix. The SDK examples use this table model, `BIDataSetTableModel`, to render datasets into the HTML result pages. The method, `renderDataset`, is provided by the `Helpers` class in the `com.sap.bi.sdk.samples` package.

API Documentation

Refer to the API documentation for the OLAP Table Model in the following package:

com.sap.ip.bi.sdk.dac.result.model

Rendering Algorithm

`BIDataSetTableModel` is constructed from `IBIDataSet`, and exposes three methods that are required for rendering a table:

- `getColumnCount()`
- `getRowCount()`
- `getValueAt(int rowIndex, int columnIndex)`

OLAP Table Model

These methods provide a projection of the data set into a two-dimensional matrix. Additional advanced data set features are supported – for example, crossjoins, dimension properties, and in particular, the hierarchical display of the data set, which can be set to on or off by the `isDisplayHierarchy` parameter in the constructor.

The strategy of the algorithm is diagrammed below:

0D SALE ORG	(20D COUNTRY)	Incoming Orders Value	Billing Value
All Sales Organizations		103,710,807.00*	184,707,824.00*
Toronto	CA	15,077,280.00 CAD	14,021,000.00 CAD
Vancouver	CA	12,224,260.00 CAD	12,016,994.00 CAD
Hamburg	DE	6,024,000.00 DM	5,852,000.00 DM
Munich	DE	21,251,500.00 DM	19,903,500.00 DM
Berlin	DE	10,759,000.00 DM	10,550,000.00 DM
Frankfurt	DE	7,144,500.00 DM	6,920,000.00 DM
Paris	FR	93,557,400.00 FRF	88,144,800.00 FRF
Birmingham	GB	4,564,700.00 £	4,453,000.00 £
London	GB	7,103,210.00 £	6,901,000.00 £
New York	US	11,849,304.00 \$	11,554,427.00 \$
San Francisco	US	4,020,053.00 \$	3,791,103.00 \$

Diagram annotations:

- Section I: Row and column labels (captions) - covers the first two columns.
- Section II: Column headers - covers the last two columns.
- Section III: Row headers - covers the first column.
- Section IV: Cell data - covers the remaining cells.
- headerCols = 2 (bracket under first two columns)
- headerRows = 1 (bracket on the right for the first row)
- totalCols = 4 (bracket under all four columns)
- totalRows = 13 (bracket on the right for all rows)

Figure 40 — OLAP Table Model Algorithm

The output table represented by this model is divided into four sections:

- I. Row and column labels (captions)
- II. Column headers
- III. Row headers
- IV. Cell data

Examples

Most of the SDK's examples are designed to deliver result sets. Examples with OLAP result sets rely on the OLAP Table Model to format multidimensional data in two dimensions. When constructing `BIDataSetTableModel` from `IBIDataSet`, the columns and rows information are retrieved from the `IBIDataSet`.

Functionality of the OLAP Table Model is delivered to the examples via the `Helpers.java` class.

Below we provide an example of how to use the OLAP Table Model, and then we explain what the `Helpers.java` class provides to the examples.



Retrieving rows information with the OLAP Table Model:

This code sample illustrates how to use the OLAP Table Model to get rows information, though the columns information is similar. It also demonstrates how you could further extend the table model interface, `BIDataSetTableModel`, for more complicated OLAP result sets.

We need the following information for the table model:

```
// number of dimensions on rows
private int _dimRows = 0;

// arrays for the count of properties for each dimension on rows
private int[] _rowPropCount = null;

// variables for the number of header rows of the output grid
private int _nHeaderRows = 0;

// these variables describe the size of the output grid and depend on the
// chosen representation
private int _nTotalRows = 0;

// these variables are used for the actual data set coordinates
private int _actualRow = 0;
```

First, get column axis cursor and row axis cursor from the data set:

```
_rows = dataset.getAxisCursor( IBIDataSet.ROWS );
```

The total rows are the sum of header rows and number of rows on the data set:

```
nTotalRows = rowNum + nHeaderRows;
```

Retrieving Result Sets

Examples

To get the number of rows on the data set:

```
_rows.last();  
rowNum = _rows.getRow();
```

For `_nHeaderRows`, there are a few additional considerations. If properties have been selected for the dimensions, these should be displayed (regardless of the axis; on rows and columns alike). In addition, hierarchy on a dimension may be displayed to an extent that multiple levels need to be taken into account.

For the calculation of the number of `nHeaderRows`, see the following documentation from the `BIDataSetTableModel` Javadoc:

```
* Row# Dimension Property  
* 1 Customer caption Mayer  
* 2 Customer Zip Code 94025  
* 3 Measures caption Sales Count
```

In this case, we have two dimensions on the columns axis: customers and measures. Customers has the property zip code selected. There are no properties selected for measures. The resulting number of header rows in this case is 3.

To account for the dimension properties, we get the number of attributes from each dimension on an axis, and accumulate the number of properties that are selected.

The header row represents the number of dimensions on the columns axis:

```
_dimCols = _cols.getDimensionCursor().size();  
_nHeaderRows = _dimCols;
```

Add the number of dimension properties for each dimension on the columns axis:

```
_nHeaderRows = _nHeaderRows + _colPropCount[dimNum];
```

The number of dimension properties can be retrieved from `DimensionAttributeCursor`, and this need to be done for every dimension on the columns axis:

Retrieving Result Sets

Examples

```
// number of properties on cols
dimNum = 0;
for (Iterator h = _cols.getDimensionCursor().iterator(); h.hasNext();) {
    IBIDimensionCursor dim = (IBIDimensionCursor) h.next();
    _colPropCount[dimNum] = dim.getDimensionAttributeCursor().size();
    _nHeaderRows = _nHeaderRows + _colPropCount[dimNum];
    dimNum++;
}
```

In this way, we calculate the number of header rows and columns.

Now, one additional complexity comes into play because of the hierarchies. First, we describe the type of graphical representation we choose to display hierarchies in the output table and how this influences our table model.

A Hierarchy on the rows axis should be displayed in the following way:

```
* #1: Caption          | #2: property 1 | ....
* All Customers
* |-----|
* | USA
* | Canada
* | |-----|
* | | Vancouver | 94025
* | | Montreal
* | France
```

Note that the separation of the different levels is done exclusively using indentation and vertical and horizontal lines in the cell UI. What's key here is that for each dimension on the rows axis we use a single column, plus the columns needed for the properties.

A Hierarchy on the columns should be displayed in the following way:

```
* | All Products |-----| #1: Caption lvl 1
* | | Food | Office | Outdoor | #2: Caption lvl 2
* | | Guido | Uwe | Jenny | #3: property 1
* |-----|
```

Using this representation, we would need one header row per visible level. The different levels are separated using a horizontal line, which is part of the cell UI.

To make the display more flexible, this model provides the option to handle the hierarchy in a different way, in which you can switch the hierarchical display completely off. In this case, one dimension on the columns axis is displayed on a single row, and one dimension on the rows axis is displayed using a single column. The variable that indicates whether the hierarchy is on or off is the Boolean `isDisplayHierarchy`:

Retrieving Result Sets

Examples

```
//apply the correction to the header rows count
if (_isDisplayingHierarchy) {
    for (int i = 0; i < _dimCols; i++)
        _nHeaderRows += _colLvlInfo[i][1];
}
```

The `_colLvlInfo` is retrieved from method `getLevelInfo` of `BIDataSetTableModel`, and its definition follows:

```
// level info for columns and rows
// [#][0] = minimum level for dimension #
// [#][1] = number of levels between minimum level and maximum level for dimension #
// [#][2] = number of levels + number of dimension properties for dimension #
// [#][3] = accumulated Header Positions
private int[][] _colLvlInfo = null;
```

When a client requests a cell using `getValueAt(int rowIndex, int colIndex)`, we ascertain to which of the four sections this cell belongs:

```
if ((rowIndex < _nHeaderRows) && (columnIndex >= _nHeaderCols)) {
    return BIDataSetTableModel.SECTION_COLUMN_HEADERS;
} else if ((columnIndex < _nHeaderCols) && (rowIndex >= _nHeaderRows)) {
    return BIDataSetTableModel.SECTION_ROW_HEADERS;
} else if ((rowIndex >= _nHeaderRows) && (columnIndex >= _nHeaderCols)) {
    return BIDataSetTableModel.SECTION_CELLS;
} else {
    return BIDataSetTableModel.SECTION_LABELS;
}
```

If it belongs to "IV: cells," the supplied `rowIndex` and `colIndex` are transformed into the actual coordinates of the data set `actualCol` and `actualRow`:

```
_actualCol = colIndex - _nHeaderCols;
_actualRow = rowIndex - _nHeaderRows;
```

The formatted value, actual value, and data type of this cell is wrapped into an object of type `BICellData`, which is further wrapped into `BITableItem`. `BITableItem` also contains section and other information for hierarchical rendering (for details, see the Javadoc for `BITableItem`):

Examples

```

_cols.absolute(_actualCol + 1);
if (_rows != null)
    _rows.absolute(_actualRow + 1);
String formattedValue = "";
Object value = null;
int type = Types.NULL;
try {
    formattedValue =
        _dataset.getString(
            IBICursorColumn.FORMATTED_VALUE.toString());
    type = _dataset.getInt(IBICursorColumn.DATATYPE.toString());
    value =
        _dataset.getObject(IBICursorColumn.VALUE.toString());
} catch (SQLException e) {
    //no cell data, but it is valid DataSet.
    formattedValue = "";
    value = "";
    type = Types.NULL;
}
return new BITableItem(
    SECTION_CELLS,
    new BICellData(formattedValue, type, value));

```

If it belongs to "II row header", the rows axis cursor is positioned to the actual coordinate of `actualRow`, and the selected dimension cursor is retrieved:

```

_rows.absolute(_actualRow + 1);
dimCursorIterator = _rows.getDimensionCursor().iterator();

//move dimensionCursor
for (int i = 0; dimCursorIterator.hasNext(); i++) {
    dim = (IBIDimensionCursor) dimCursorIterator.next();

    if (i == _colInfo[colIndex][0]) {
        break;
    }
}

```

The column header information is collected into `_colInfo` via method `setHeaderPositionInfo` of `BIDataSetTableModel`, and its definition follows:

```

// rowInfo contains information to what dimension a column header row belongs
// and what property of that dimension it displays
// 0 = caption
// # = property with index #+1
// these arrays contain two pieces of information
// I : colsInfo[#][0] = dimension number on the axis
// II : colsInfo[#][1] = caption or property for the dimension specified in 1
private int[][] _rowInfo = null;

```

Examples

If the selected cell is a dimension member, the `IBIDimensionCursor` Object is wrapped into the `BITableItem` Object. If the selected cell is a dimension property, the property name is retrieved from the `DimensionAttributeCursor` and wrapped into the `BITableItem` Object:

```
switch (_colInfo[colIndex][1]) {
    //member caption
    case -1 :
        return new BITableItem(
            SECTION_ROW_HEADERS,
            dim,
            _rowLvlInfo[_colInfo[colIndex][0]));
    //member properties
    default :
        List rc = dim.getDimensionAttributeCursor();
        Iterator rci = rc.iterator();
        IBIDimensionAttributeCursor ac = null;
        //move dimensionAttributeCursor
        for (int j = 0; rci.hasNext(); j++) {
            ac = (IBIDimensionAttributeCursor) rci.next();

            if (j == _colInfo[colIndex][1])
                break;
        }
        // output dimension attributes
        return new BITableItem(
            SECTION_ROW_HEADERS,
            ac.getString(IBCursorColumn.NAME.toString()),
            _rowLvlInfo[_colInfo[colIndex][0]));
}
```



Helpers.java – OLAP Table Model functionality:

An example of how to use the OLAP Table Model is shown in the helper method `renderDataSet` in `Helpers.java`. The code is straightforward and self-explanatory. It simply uses a nested loop over all rows and columns, and outputs the string value returned by `getValueAt` for each cell of the projection. Note the special handling in the code for the hierarchical display.



Helpers.java – HTML stylesheet:

`Helpers.java` also helps you format result sets into HTML tables by providing an HTML doctype definition and a stylesheet. The examples use this to format tables with distinct styles for table headers and striped cell data rows.



Note:

See Appendix B: Examples for the full index of examples and instructions on getting your system up and running with them.

Chapter 6: Exceptions

Overview

This chapter describes the SDK's exception framework, which provides classes and interfaces for handling exceptions thrown by the SDK layer.

API Documentation:

Refer to the Javadocs for the exception framework in the following package of the SDK:

com.sap.ip.bi.sdk.exception

Exception Handling

All exceptions thrown by the SDK interfaces implement the `IBaseException` interface in order to comply with SAP solution production standards. This also provides integration with SAP's Java logging and tracing framework used for instrumenting the code in case of error situations.

The SDK exceptions either directly implement the `IBaseException` interface, as in the case of `BIResourceException`, or they extend exceptions in the package `com.sap.exception`, as with `BaseException`.

Legacy exceptions, which are exceptions that are thrown by components outside the SDK, do not implement the `IBaseException` interface but are handled by providing wrapper exceptions. These wrappers have exactly the same semantics as the underlying legacy exception and simply add the implementation of `IBaseException`. For example, the legacy exception `BIResourceException` wraps exceptions of the type `javax.resource.ResourceException`.

The diagram below illustrates how the BI Java SDK exceptions relate to legacy exceptions such as `javax.resource.ResourceException` and the SAP Exception Framework base exceptions:

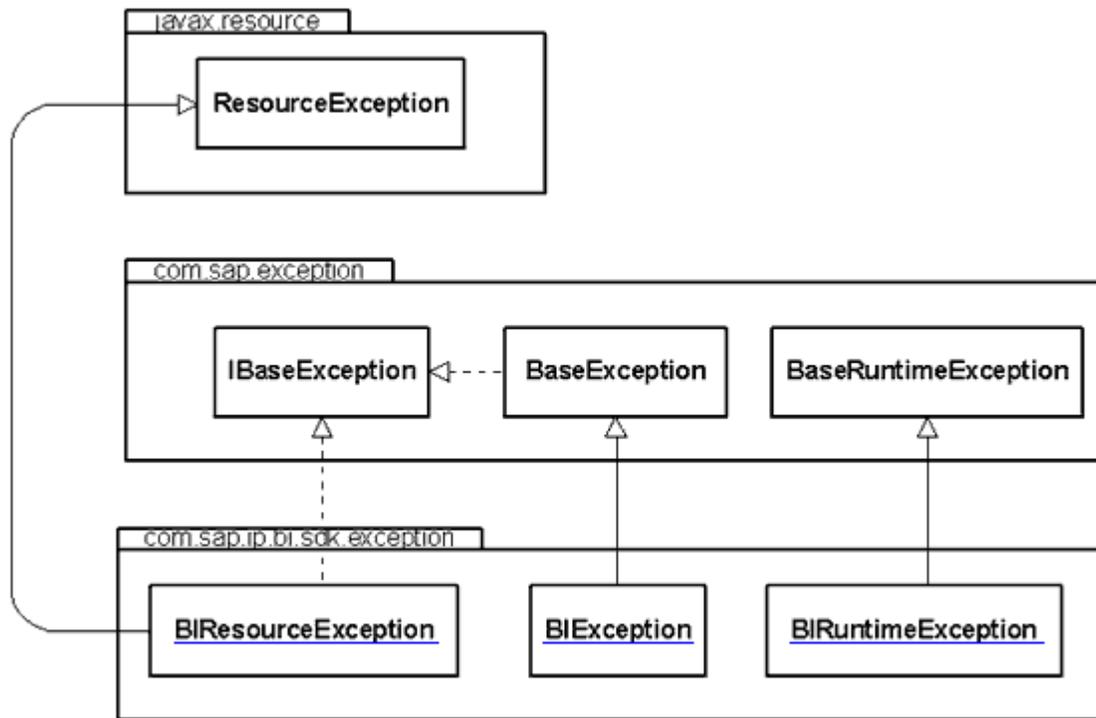


Figure 41 — BI Java SDK Exception Framework

 **Note:**

For general information on writing exceptions, see Sun’s exceptions tutorial:
<http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html>

Exception Translation

Exceptions evoked on the BI Java SDK layer are localized into all languages supported by SAP BI. To set the language for SDK exceptions, use the language property of your BI Java Connector in your connection specification. For the list of languages supported by SAP BI, see the following link in the SAP Service Marketplace:
http://service.sap.com/~form/sapnet?_SHORTKEY=01100035870000523354&

See BI Java Connectors in Appendix A: Installation for more information about the connectors and their connection properties.

Appendix A: Installation

Overview

This appendix describes the installation process and requirements for the BI Java SDK and briefly introduces the BI Java Connectors, in the following sections:

- System Requirements
- Classpath Configuration
- Logging and Tracing, JARM
- Using the BI XMLA Connector in a non-managed environment
- Documentation
- BI Java Connectors



Tip:

Be sure to check the release notes for information about new or enhanced functionality and bugs addressed between releases of the BI Java SDK. These notes are located in the docs folder in the unzipped SDK archive, in [release_notes.html](#); or simply choose Release Notes from the menu in the home page for the SDK documentation set (index.html in the docs folder of the unzipped archive).

System Requirements

See the table below for the key system requirements for the BI Java SDK:

System requirements and details

Requirement	Details
Operating system	Developing with the BI ODBO Connector requires Windows NT/2000/XP. Otherwise, there are no operating system restrictions.
Java Development Kit	1.4+
Web browser	To view this documentation set, including the Javadocs, we recommend Internet Explorer 5 or Netscape 6 and above, or compatible frames-capable browsers such as Mozilla, Opera 7+, and Safari.

Requirement	Details
Adobe Acrobat Viewer	To view the Developer's Guide (PDF), Adobe's Acrobat Viewer is required.
Memory	Recommended: at least 256MB
Disk space	Recommended: at least 50MB
Processor	Recommended: at least 300MHz
JDBC	If you are developing with the BI JDBC connector, you need to ensure that the JAR files of your database provider's JDBC driver are available in your classpath.
ODBO	If you are developing with the BI ODBO Connector, copy the JNI library <code>sapbiado.dll</code> from the <code>lib</code> folder into a location in your system path (for example, into your <code>WINNT\system32</code> directory).
SAP Query	If you are developing with the BI SAP Query Connector, you need to be sure your system is properly configured with SAP JCo. JCo is available on the SAP Service Marketplace at: http://service.sap.com/connectors/ → SAP Java Connector → Tools & Services. We recommend version 2.1.x.
XMLA	If you are developing with the BI XMLA Connector, no additional configuration is necessary. See Using the BI XMLA Connector in a non-managed environment, below.
BI Java Connector	To deploy applications created with the SDK onto the J2EE server, you need the appropriate BI Java Connector (see BI Java Connectors, below).

Classpath Configuration

Configure your CLASSPATH for the SDK by adding the SDK's libraries. These are located in the `/lib/` folder in the folder in which you have unpackaged the SDK. This folder contains all the libraries required by the SDK for development.

Logging and Tracing, JARM

Optionally, you may configure instrumentation for logging and tracing, and JARM. See the `util` package in the API documentation for instructions on using methods together with properties files to activate these features in a non-managed environment.

API Documentation:

For code samples, and for more information, see the instructions in the package documentation for the SDK's `util` package:

com.sap.ip.bi.sdk.util

Using the BI XMLA Connector in a non-managed environment

Since the BI XMLA Connector relies upon the SAP XML Parser package (`SAPXMLToolkit.jar`) and SAP SOAP package (`webservices_lib.jar`) normally present in a managed (J2EE) environment, some guidelines are helpful when using the XMLA connector in a non-managed environment (in the absence of SAP's J2EE server). In this case, you need to manually configure the correct reference to SAP's XML Parser and SOAP packages in the host application.

We provide some guidelines below on how to configure the correct parser reference. Please note that these guidelines are intended just for reference, and don't address, for example, scenarios where an application defines its own class loading mechanisms or refers to a default XML parser. In those cases, please consult the documentation of the host application on how to configure a reference to the third party XML Parser (in this case, the SAP XML Parser and SAP SOAP packages).

To manually configure your parser reference:

1. Ensure that the SAP XML Parser and SAP SOAP packages (`SAPXMLToolkit.jar` and `webservices_lib.jar`) are in your classpath.
2. If there are other XML parser packages in the classpath, move them behind the SAP XML Parser and SAP SOAP packages in your classpath sequence, or remove them if possible.
3. Configure your system property. For example, use the following code to set the reference to SAP packages:

```
System.setProperty("javax.xml.parsers.SAXParserFactory",  
"com.sap.engine.lib.jaxp.SAXParserFactoryImpl");  
System.setProperty("javax.xml.soap.MessageFactory",  
"com.sap.engine.services.webservices.jaxm.soap.MessageFactoryImpl");  
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",  
"com.sap.engine.lib.jaxp.DocumentBuilderFactory");  
System.setProperty("javax.xml.transform.TransformerFactory",  
"com.sap.engine.lib.jaxp.TransformerFactoryImpl");  
System.setProperty("javax.xml.soap.SOAPConnectionFactory",  
"com.sap.engine.services.webservices.jaxm.soap.SOAPConnectionFactoryImpl");
```

Use this option with caution, however, since the system property might affect other code, especially if the other code is using different XML Parser.

Some common error messages that may appear when the parser package is not configured correctly:

```
class not found exception:  
com.sap.engine.services.webservices.jaxm.soap.MessageFactoryImpl not found  
  
class not found exception: org.apache.crimson.jaxp.DocumentBuilderFactoryImpl not  
found  
  
javax.xml.parsers.FactoryConfigurationError: Provider  
org.apache.crimson.jaxp.DocumentBuilderFactoryImpl not found at  
javax.xml.parsers.DocumentBuilderFactory.newInstance( DocumentBuilderFactory.java:109)
```

Documentation

In addition to this Guide, the BI Java SDK package provides Javadocs and additional documentation in HTML format. As an entry point, use `index.html` in the `docs` folder of the folder in which you unzipped the SDK archive:

```
<installation drive>/<installation directory>/docs/index.html
```



In many cases, this Guide provides links to other components of the documentation set, for example, to Javadoc packages, so that you can easily navigate to them. These links are set relative to the original location of the Guide, in the `docs/devguide` folder of the distribution package, and will not resolve if the `devguide.pdf` file is removed from this folder.

The links have been tested on Windows systems only, and may not work on other systems.

How-To Guides

Three How-To Guides ship in the BI Java SDK documentation set, and you can access these from the Documentation page in the SDK package. The How-To Guides include:

- **How To Use the BI Java SDK in a Portal iView**
This document provides detailed instructions on how to use the BI Java SDK and its BI Java Connectors in an Portal iView. It contains step-by-step instructions for creating an iView which uses the BI Java SDK's BI XMLA Connector to connect to an SAP BI system and retrieve a list of schemas.

BI Java Connectors

- **How To Use the BI Java SDK in a J2EE Application**

This document provides detailed instructions on how to use the BI Java SDK and its BI Java Connectors in a J2EE application. It contains step-by-step instructions for creating a servlet that uses the BI XMLA Connector to connect to an SAP BI system.

- **How To Use the BI Java SDK in a Web Dynpro Application**

This document provides detailed instructions on how to use the BI Java SDK and its BI Java Connectors in a Web Dynpro Application. It contains step-by-step instructions for adding the required library references to use the BI Java SDK, and for establishing a connection using one of the BI Java Connectors in the Web Dynpro and J2EE environment.

BI Java Connectors

The BI Java Connectors are a group of four JCA (J2EE Connector Architecture)-compliant resource adapters that implement the BI Java SDK's APIs and allow you to connect the applications you build with the SDK to heterogeneous data sources. The BI Java Connectors may be deployed onto SAP NetWeaver '04 - Web Application Server version 6.40.

The BI Java SDK contains the JAR files you need to develop applications using any of the BI Java Connectors and to use them in an unmanaged scenario, but to use your application with a data source in the managed environment of the J2EE server, you need to deploy the appropriate BI Java Connector.



Note:

The BI Java Connectors are distributed separately from the BI Java SDK, deployed with NetWeaver's optional BI UDI component.

The BI Java Connectors are packaged in resource adapter archives, or RAR files. Each RAR file includes class libraries and dependencies, a deployment descriptor, and connector properties documentation in the form of a `howto.html` file. General system guidelines are listed below in the Connector Overview section.



Note:

The connectors' `howto.html` files are also included in the SDK distribution package for your reference. See [index.html](#) in the docs folder of the package, then select Connectors.

Four BI Java Connectors are available, listed below with the name of the resource adapter archive in which they are deployed:

- BI JDBC Connector : `bi_sdk_jdbc.rar`
- BI ODBO Connector : `bi_sdk_odbo.rar`
- BI SAP Query Connector : `bi_sdk_sapq.rar`

BI Java Connectors

- BI XMLA Connector : `bi_sdk_xmla.rar` or `bi_sdk_xmla_proxy.rar` (see Note in BI Java Connectors Overview, below)

BI Java Connectors Overview

The following table provides an overview of the BI Java Connectors:

BI Java Connectors and details

Connector	Access to	Technology based on	System requirements
BI JDBC Connector	Relational data sources: over 170 JDBC drivers Examples: Teradata, Oracle, Microsoft SQL Server, Microsoft Access, DB2, Microsoft Excel, text files such as CSV	Sun's JDBC (Java Database Connectivity) – the standard Java API for relational database management systems (RDBMS).	JDBC driver
BI ODBO Connector	OLAP data sources: ODBO-compliant data sources Examples: Microsoft Analysis Services, SAS, Microsoft PivotTable Services	Microsoft's ODBO (OLE DB for OLAP) – the established industry-standard OLAP API for the Windows platform.	Microsoft Windows 2000 / NT / XP OLE DB for OLAP (ODBO) driver for your data source
BI SAP Query Connector	SAP operational applications Examples: data in transactional systems such as R/3, Ad-Hoc, and Operational Reporting	SAP Query – a component of SAP's Web Application Server that allows you to create custom reports without any ABAP programming knowledge.	SAP JCo
BI XMLA Connector	OLAP data sources Examples: MS Analysis Services, Hyperion, MicroStrategy, and BW 3.x	Microsoft's XMLA (XML for Analysis) – Web services-based, platform-independent access to OLAP providers. Exchanges analytical data between a client application and a data provider working over the Web, using a SOAP-based XML communication API.	none

Testing the BI Java Connectors

We provide servlets with which to test the configuration of the BI Java Connectors. For usage information, see Testing the Connections in the connector package documentation in the Javadocs at:

com.sap.ip.bi.sdk.dac.connector

For Additional Information

- For connector properties configuration information, refer to the `howto.html` file inside the resource adapter archive (RAR file). The `howto.html` files for each connector also included in the SDK distribution package for your reference. See [index.html](#) in the docs folder of the package, then select Connectors.
- For the SDK's connection architecture API documentation, refer to the Connection Interfaces Javadoc package, at: com.sap.ip.bi.sdk.dac.connector.
- For more on the SDK's connection architecture, see Connection Architecture.

Appendix B: Examples

Overview

The BI Java SDK package provides the source code for several example servlets that are described throughout this Guide. This appendix introduces the examples and helps get you up and running, in the following sections:

- Finding the examples
- Configuring your system
- Index of examples

Finding the Examples

You can find the full source code of the examples in the following path after unzipping the BI Java SDK distribution archive:

```
<installation drive>/<installation directory>/docs/examples/
```

The files are also linked from the included HTML documentation, starting at *index.html* in the docs folder of the unzipped archive; choose the Examples link.

The filenames of the examples are constructed according to the following naming conventions:

- Tutorial_*.java → examples used in the Getting Started tutorials
- Olap_*.java → examples for working with OLAP data sources
- Relational_*.java → examples for working with relational data sources

In the above, * simply iterates the number of the example.

Along with the example Java source code, we provide corresponding HTML files which render the source code in HTML format for easy visibility. These files are named as follows:

```
*.java.html
```

In the above, * is the name of the example.

Configuring your System

In addition, we provide the rendered result of the servlet in HTML format:

```
*_1.result.html
```

Again, * is the name of the example.



Tip:

Although most of our examples are servlets that render the data sets and additional explanatory information into an HTML stream, the SDK libraries can be used in other scenarios, such as to build standalone Java applications.

Configuring your System

First, be sure you've consulted the SDK installation instructions in Appendix A: Installation, and then follow with the sections below to get up and running with the examples.

Data Sources

You can view the example source code and HTML results, but to actually work with and run the examples, you need the following data sources:

- **OLAP examples:**
SAP BI system, release 2.0 or higher, with the SAP demo InfoCube *SAP Demo Sales and Distribution: Overview*, and the query *OD_SD_C03/OD_SD_C03_Q009 Order and Sales values* activated.
- **Relational examples:**
An active JDBC data source for which you have a valid user name and password, with its JDBC drivers properly configured in your classpath.



Note:

The relational examples are configured to work with a JDBC database present on your own system. The output we include in the documentation set shows results against just one example JDBC database, so your results will vary.

Rendering to File

The examples implement a minimal HTTP servlet, which generates HTML for easy viewing of results. By default, running the `main` method without a parameter will write the HTML to the console. To write it to an HTML file instead, specify a filename, with full path and `.html` extension, as the parameter.

Connection Properties

Four properties files, one for each BI Java Connector, provide connection properties to the connectors and are included in the examples folder, nested in the `com.sap.ip.bi.sdk.samples` package with the Java source files:

- `Helpers.jdbc.properties`: connection information for the BI JDBC Connector
- `Helpers.odbo.properties`: connection information for the BI ODBO Connector
- `Helpers.sapq.properties`: connection information for the BI SAP Query Connector
- `Helpers.xmla.properties`: connection information for the BI XMLA Connector

Edit the existing properties, or create new files to locally override the properties, named `Helpers.nnnn.local.properties` (where "nnnn" corresponds to the four-letter connector name). The examples will first look for the local file, and if not found, will take the original properties files.

For connection property configuration information, refer to the `howto.html` file that ships inside of each resource adapter archive. The `howto.html` files for each connector are also included in the SDK distribution package for your reference. See [index.html](#) in the docs folder of the package, then select Connectors.

Index of Examples

The following table provides an overview of the examples included in the BI Java SDK package:

BI Java SDK examples

Example	Description
Tutorial_1.java	Getting Started - OLAP Tutorial Contains a complete end-to-end scenario, demonstrating how to connect to an SAP BI system using the BI XMLA Connector, retrieve a cube from the data source, create a query, execute it, and render the result set into an HTML table.
Tutorial_2.java	Getting Started - Relational Tutorial Contains a complete end-to-end scenario, demonstrating how to connect to a JDBC database using the BI JDBC Connector, retrieve a table from the data

Example	Description
	source, create a query, execute it, and render the result set into an HTML table.
Olap_1.java	<p>OLAP 1 - Accessing OLAP metadata</p> <p>Demonstrates four different ways to retrieve OLAP metadata:</p> <ol style="list-style-type: none"> 1. Via connection-level methods 2. Via <code>ObjectFinder</code> methods 3. Via JMI methods 4. Via member data access methods
Olap_2.java	<p>OLAP 2 - Direct execution of MDX statement</p> <p>Demonstrates how to retrieve a result set by directly executing an MDX statement, then shows how to display the result set as an HTML table.</p>
Olap_3.java	<p>OLAP 3 - Pivoting / changing layout of an OLAP query</p> <p>Illustrates the process of changing the layout of a query by moving dimensions between axes and then by swapping axes. Renders the output of each into two separate HTML tables.</p>
Olap_4.java	<p>OLAP 4 - Selecting dimension attributes</p> <p>Selects a dimension attribute, and renders the result set into an HTML table.</p>
Olap_5.java	<p>OLAP 5 - Sorting by measure value</p> <p>Renders the default result set into an HTML table, sorts the data according to a measure value in ascending order, and then renders the data into a second HTML table for comparison.</p>
Olap_6.java	<p>OLAP 6 - Sorting by dimension attribute</p> <p>Illustrates how to select a dimension attribute for display, and to sort by a dimension attribute.</p>
Olap_7.java	<p>OLAP 7 - Filtering</p> <p>Illustrates both a ranking filter and a condition-based filter. Renders the result of a query without any filtering, then filters the set of Sold-To parties using a ranking filter and re-renders the result. Changes the filter to a condition-based filter to restrict by quantity, and re-renders the result for comparison.</p>
Olap_8.java	<p>OLAP 8 - Hierarchy navigation - member drill operations</p> <p>Illustrates hierarchy navigation by applying the following operations in sequence to an initial data set:</p> <ol style="list-style-type: none"> 1. Zoom in 2. Zoom out 3. Drill down 4. Drill up <p>After each operation, the result set is rendered again for comparison.</p>
Olap_9.java	<p>OLAP 9 - Calculated members</p> <p>Creates a calculated measure - cost per item - by dividing the total cost by the</p>

Example	Description
	number of items sold.
Olap_10.java	<p>OLAP 10 - SAP variable selection and editing</p> <p>Illustrates retrieval of and effect of editing an SAP variable. Sets a value for an optional SAP variable, renders the result of the default query into a table, searches for a specific optional SAP variable using the OLAP object finder, changes its value, and then re-renders the result into a new table for comparison.</p>
Relational_1.java	<p>Relational 1 - Accessing relational metadata</p> <p>Illustrates the process of retrieving relational metadata from catalog to column from a JDBC data source.</p>
Relational_2.java	<p>Relational 2 - Accessing relational metadata: 2</p> <p>Demonstrates three different ways to retrieve relational metadata:</p> <ol style="list-style-type: none"> 1. Via connection-level methods 2. Via <code>ObjectFinder</code> methods 3. Via JMI methods
Relational_3.java	<p>Relational 3 - Direct execution of SQL statement</p> <p>Demonstrates how to retrieve a result set by directly executing a SQL statement, then shows how to display the result set as an HTML table.</p>
Relational_4.java	<p>Relational 4 - Simple relational query</p> <p>Demonstrates how to retrieve a result set by creating a simple query, then shows how to display the result set as an HTML table.</p>
Relational_5.java	<p>Relational 5 - More complex relational query</p> <p>Demonstrates how to retrieve a result set by creating a more complex query with the following features:</p> <ol style="list-style-type: none"> 1. Field selections 2. Joins 3. Sorting
Helpers.java	<p>Helper methods</p> <p>Provides static helper methods that facilitate connecting to data sources and rendering result sets.</p>

Appendix C: Additional Resources

Web References

SAP JCo and JCo-OS specific downloads:

- <http://service.sap.com/connectors>

SAP Query:

- http://help.sap.com/saphelp_nw04/helpdata/en/d2/CB3EFB455611D189710000E8322D00/frameset.htm

SAP UD Connect:

- http://help.sap.com/saphelp_nw04/helpdata/en/78/EF1441A509064ABEE6FFD6F38278FD/frameset.htm

SAP Web Application Server's Visual Administrator:

- http://help.sap.com/saphelp_nw04/helpdata/en/39/83682615CD4F8197D0612529F2165F/frameset.htm

SAP Web Application Server's Logging API:

- http://help.sap.com/saphelp_nw04/helpdata/en/4A/C3953FF1353C17E1000000A114084/frameset.htm

SAP's chapters on SAP Variables:

- http://help.sap.com/saphelp_nw04/helpdata/en/E2/17533D6DD60610E1000000A114084/frameset.htm

Sun's JDBC:

- JDBC Technology – <http://java.sun.com/products/jdbc/index.html>
- JDBC Drivers – <http://industry.java.sun.com/products/jdbc/drivers>

Sun's JCA:

- Links to the specification and other downloads – <http://java.sun.com/j2ee/connector/>.
- JCA Specification – <http://www.jcp.org/en/jsr/detail?id=16>

Sun's exceptions tutorial:

- <http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html>

XMLA and related topics:

- XMLA Council – <http://www.xmla.org/>

Additional Resources

Web References

- World Wide Web Consortium (W3C) specification for SOAP – <http://www.w3.org/TR/SOAP/>
- W3C information on Extensible Markup Language (XML) – <http://www.w3.org/XML/>
- Microsoft on XMLA – <http://msdn.microsoft.com/library/default.asp?URL=/library/techart/XMLAnalysis.htm>

OMG's CWM and MOF standards:

- <http://www.omg.org/cwm/>.

MOF 1.3 specification:

- <http://www.omg.org/cgi-bin/doc?formal/00-04-03.pdf>

XMI:

- <http://www.omg.org/technology/documents/formal/xmi.htm>

JMI – the 1.0 release of the JMI specification:

- <http://jcp.org/jsr/detail/40.jsp>.

Microsoft's documentation on MDX:

- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/olapdmad/agmdxbasics_04qg.asp?frame=true

Microsoft's "Comparison of SQL and MDX":

- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/olapdmad/agmdxbasics_90qg.asp

Microsoft's Data Access Components (MDAC) SDK Overview:

- <http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/dasdk/mdac3sc7.htm>

Microsoft Universal Data Access (UDA) – OLE DB, ADO and XML for Analysis are part of Microsoft's Universal Data Access Architecture. See <http://www.microsoft.com/data/default.htm> as the general entry point into Microsoft UDA.

Books

Books

Kimball, Ralph. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. Wiley Computer Publishing, John Wiley & Sons, Inc., 1996.

SAP BW Reporting Made Easy, Release 2.0B/2.1C. Palo Alto: SAP Labs, Inc., 2001.

Gamma, Erich; et. al. *Design Patterns*. Addison-Wesley Pub Co; 1st edition (January 15, 1995)

Appendix D: Glossary

ActiveX Data Objects (ADO)

A logical object model for programmatically accessing a variety of data sources through OLE DB interfaces. ADO, provided by Microsoft, is the foundation for Microsoft's ADO MD extension, upon which the BI ODBO Connector is based.

ActiveX Data Objects Multidimensional (ADO MD)

A logical object model provided by Microsoft that facilitates easy access to multidimensional data by extending ADO with objects specific to multidimensional data, such as cubes and cellsets. Like ADO, ADO MD uses an underlying OLE DB provider to gain access to data. The BI ODBO Connector uses ADO MD to support connectivity to OLAP data sources.

ADO

Acronym for ActiveX Data Objects.

ADO MD

Acronym for ActiveX Data Objects Multidimensional.

BI Java Connector

One of a set of four JCA (J2EE Connector Architecture)-compliant resource adapters that allow you to connect applications built with the BI Java SDK to heterogeneous data sources:

- BI JDBC Connector (for relational JDBC-compliant data sources)
- BI ODBO Connector (for ODBO-compliant OLAP data sources)
- BI SAP Query Connector (a component of the SAP Web Application Server Basis)
- BI XMLA Connector (for OLAP data sources such as SAP BW 3.x)

You can also use the Connectors to make external data sources available in SAP BI, via SAP BI's UD Connect. In the SDK, the term connector is synonymous with resource adapter.

BI Java SDK

BI Java SDK

Abbreviation for SAP's Business Intelligence Java Software Development Kit.

BI JDBC Connector

A resource adapter for the Business Intelligence domain based on Sun's Java Database Connectivity (JDBC), which is the standard Java API for relational database management systems (RDBMS). The BI JDBC Connector may be deployed into SAP's Web Application Server, and allows you to connect applications built with the BI Java SDK to over 170 JDBC drivers, supporting data sources such as Teradata, Oracle, Microsoft SQL Server, Microsoft Access, DB2, Microsoft Excel, and text files such as CSV.

You can also use the BI JDBC Connector to make these data sources available in SAP BI, via SAP BI's UD Connect.

The JDBC Connector implements the BI Java SDK's `IBIRelational` interface.

BI ODBO Connector

A resource adapter for the Business Intelligence domain based on Microsoft's OLE DB for OLAP (ODBO), which is the established industry-standard OLAP API for the Windows platform. The BI ODBO Connector may be deployed into SAP's Web Application Server, and allows you to connect applications built with the BI Java SDK to ODBO-compliant OLAP data sources such as Microsoft Analysis Services, SAS, and Microsoft PivotTable Services.

You can also use the BI ODBO Connector to make these data sources available in SAP BI, via SAP BI's UD Connect.

The ODBO Connector implements the BI Java SDK's `IBIOlap` interface.

BI SAP Query Connector

A resource adapter for the Business Intelligence domain based on SAP Query, which is a component of SAP's Web Application Server that allows you to create custom reports without any ABAP programming knowledge. The BI SAP Query Connector uses SAP Query to allow applications created with the BI Java SDK to access data from these SAP operational applications.

You can also use the BI SAP Query Connector to make these data sources available in SAP BI, via SAP BI's UD Connect.

The SAP Query Connector implements the BI Java SDK's `IBIRelational` interface.

BI XMLA Connector

A resource adapter for the Business Intelligence domain based on Microsoft's XML for Analysis (XMLA), that facilitates Web services-based, platform-independent access to OLAP providers. The BI XMLA Connector may be deployed into SAP's Web Application Server, and enables the exchange of analytical data between a client application and a data provider working over the Web, using a SOAP-based XML communication API.

Business Intelligence Java Software Development Kit (BI Java SDK)

The BI XMLA Connector allows you to connect applications built with the BI Java SDK to data sources such as Microsoft Analysis Services, Hyperion, MicroStrategy, MIS, and BW 3.x.

You can also use the BI XMLA Connector to make these data sources available in SAP BI, via SAP BI's UD Connect.

The BI XMLA Connector implements the BI Java SDK's `IBIOlap` interface.

Business Intelligence Java Software Development Kit (BI Java SDK)

A Java software development kit with which you can build analytical applications that access, manipulate, and display both multidimensional (Online Analytical Processing, or OLAP) and tabular (relational) data. The BI Java SDK consists of:

- Java APIs for accessing, manipulating, and displaying data from diverse data sources
- Documentation
- Examples

column

An element of a table that describes its structure and the types of its rows. A column has a name, a data type, and an implicit order (ordinal) based on the order chosen when defining the table. Columns can also belong to indexes, which are used to ensure uniqueness (set property) of rows in the table.

command processor

Part of each of the BI Java SDK's query APIs, interfaces that make it easier to use the underlying query models by hiding the complexity of these models. With the command processors, you can create and manipulate complex queries with simple commands. You can think of the individual methods of the command processors in terms of macros that consist of several method calls manipulating the structures of queries.

The SDK provides two command processors:

- OLAP Command Processor, for manipulating OLAP queries
- Relational Command Processor, for manipulating relational queries

Common Client Interface (CCI)

An API defined by Sun's JCA specification that is common across heterogeneous EISs. It is designed to be "toolable" – that is, it leverages the Java Beans architecture so that development tools can incorporate the CCI into their architecture.

Note that the BI Java Connectors implement only the connection interfaces defined by the CCI. The CCI's interaction interfaces, data interfaces, and metadata interfaces, however, are not implemented by the BI Java SDK. BI-specific client APIs that are tailored for OLAP interactions are provided by the BI Java Connectors.

Common Warehouse Metamodel (CWM)

Common Warehouse Metamodel (CWM)

An Object Management Group (OMG) standard that provides for a common understanding of metadata in order to exchange it between heterogeneous systems. CWM describes the exchange of metadata in the data warehousing and analysis, business intelligence, knowledge management, and portal technologies domains. CWM is MOF is the modeling language for CWM, UML is its modeling notation, and XMI is used to interchange the metadata.

CWM is capable of modeling a wide spectrum of OLAP and relational providers. The SDK uses CWM to represent relational and OLAP data in the Relational and OLAP Metadata Models.

For more information about CWM, see <http://www.omg.org/cwm/>.

connector

Synonym for resource adapter. See more at BI Java Connector.

Connector Gateway

An SAP NetWeaver Portal service that provides instances of connections to Portal components.

cube

In the OLAP domain, set of data organized as a multidimensional structure defined according to dimensions and measures. Related SAP BI concepts include InfoCube and query.

CWM

Acronym for Common Warehouse Metamodel.

deployment descriptor

Assists in deploying a resource adapter (known as connector in the SDK) by defining the contract between a resource adapter provider and a deployer. The deployment descriptor file contains information about which classes implement the interfaces with which the application server interacts.

dimension

In the OLAP domain, a collection of similar data which, together with other such collections, forms the structure of a cube. Typical dimensions include time, product, and geography. Each dimension may be organized into a basic

Enterprise Information System (EIS)

parent-child hierarchy or, if supported by the data source, a hierarchy of levels. For example, a geography dimension might include levels for continent, country, state, and city.

Note that in SAP BI, InfoObject (characteristic) is a related term, but "dimension" means something entirely different in SAP BI than it does in the OLAP domain.

Enterprise Information System (EIS)

A system such as an ERP (Enterprise Resource Planning), database, or mainframe transaction processing system which forms the information infrastructure of an enterprise system.

filter

A set of criteria that restricts the set of records returned as the result of a query. With filters, you define which subset of data appears in the result set.

hierarchy

A logical tree structure that organizes the members of a dimension into a parent-child relationship. If supported by the data source, the hierarchy consists of levels, where the top level is an aggregate of all members and each subsequent level has zero or more child members.

IBIOlap

An interface provided by the BI Java SDK and implemented by all OLAP connectors which serves as an entry point to interfaces that support access to multidimensional metadata and queries.

IBIRelational

An interface provided by the BI Java SDK and implemented by all relational connectors which serves as a point of entry to a set of interfaces that provide access to relational metadata and queries.

INative

An optional interface defined in the Portal Connection Framework API which can be implemented by a connector. INative enables you to access the connected EIS via an API that is tailored specifically for that underlying EIS. The interface returned depends on the connected EIS.

J2EE Connector Architecture (JCA)

J2EE Connector Architecture (JCA)

A standard architecture from Sun designed for connecting J2EE servers with EISs. The architecture defines a set of contracts, such as transactions, security, and connection management, that a connector has to support to plug in to an application server.

JCA provides an API for connecting to heterogeneous data sources in a consistent manner. The BI Java Connectors are JCA-compliant.

Java Metadata Interface (JMI)

An extensible metadata service for the Java platform that provides a common Java programming model for accessing metadata. JMI defines a Java mapping for the Meta Object Facility (MOF) specification from the Object Management Group (OMG). The SDK uses JMI mapping to render its query and metadata models into Java APIs.

For more information, see <http://jcp.org/jsr/detail/40.jsp>.

JCA

Acronym for J2EE Connector Architecture

JCo (SAP Java Connector)

SAP's toolkit that allows a Java application to communicate with any SAP system. It is used by the BI SAP Query Connector to interact with SAP Web Application Server instances.

For more information about JCo, visit the SAP Service Marketplace at: <http://service.sap.com/connectors/>

JDBC (Java Database Connectivity)

Provides an API that lets you access relational databases using the Java programming language. It provides cross-DBMS connectivity to a wide range of SQL databases, and also provides access to tabular data sources such as spreadsheets or flat files.

For more information, see <http://java.sun.com/products/jdbc/index.html>.

JMI

Acronym for Java Metadata Interface.

JMI service (Java Metadata Interface Service)

JMI service (Java Metadata Interface Service)

Any system that provides a JMI-compliant API to access its public metadata. The BI Java Connectors expose metadata of the underlying EIS via JMI services.

level

A set of nodes (members) in a tree hierarchy in supporting data sources that are at the same distance from the root of the tree. For example, in a geography hierarchy, the top level might be all places, the second level might be continents, the third level might be countries, and the fourth level might be cities.

MDX (Multidimensional Expressions)

Microsoft's SQL-like query language used to retrieve and manipulate multidimensional data.

measure

One category of values – usually numeric – used to define a cube. These values are derived from one or more columns in the cube's fact table and are the basis for aggregation and analysis. In SAP BI, related terms include key figure and structure element.

member

An element of a dimension that represents one or more occurrences of data. A member can be unique (it occurs only once) or non-unique (it may occur more than once in its dimension). For example, in a geography dimension that includes cities in the US, the member Portland could be non-unique, since there is a city called Portland in the state of Oregon and in the state of Maine.

In SAP BI, members are referred to as instances of characteristics.

Meta Object Facility (MOF)

An OMG (Object Management Group) standard for the specification of interoperable metamodels. MOF defines language rules (syntax and semantics) for constructing metamodels and provides programming tools for saving and accessing metadata in repositories. The MOF standard is integrated in XMI, and CWM uses MOF as its modeling language.

MOF can also refer to any metadata service which abides by the MOF specifications. CWM is a MOF-compliant metamodel.

metadata API

metadata API

A set of interfaces provided by the BI Java SDK which expose the metadata of a given data source. The SDK includes two metadata APIs, both generated via JMI from their respective metadata models:

- OLAP Metadata API, for exposing metadata in an OLAP data source
- Relational Metadata API, for exposing metadata in a relational data source

metadata model

An abstract language for expressing metadata. The BI Java SDK leverages CWM metadata models (metamodels), and the following two CWM packages in particular:

- `org.omg.cwm.analysis.olap` → basis of the SDK's OLAP Metadata Model, for expressing the metadata of a multidimensional data source
- `org.omg.cwm.resource.relational` → basis of the SDK's Relational Metadata Model, for expressing the metadata of a relational data source

The SDK's metadata models also rely upon reference classes from CWM's Foundation and Objectmodel layers.

metadata repository

Contains the different classes of metadata and is capable of persisting and retrieving MOF-compliant objects, resulting in a consistent and homogeneous data model across all source systems. Metadata repository is a general term used by the Object Management Group (OMG). The Metamodel Repository is SAP's implementation of the metadata repository.

Metamodel Repository (MMR)

SAP's implementation of a metamodel and metadata repository. It is named after the metamodel layer (the meta-meta-data, or m2 layer) of the OMG's Meta Object Facility (MOF), which is the main focus in SAP BI.

MOF

Acronym for Meta Object Facility

multidimensional data

Data in dimensional models suitable for business analytics. In this documentation, we use the term "multidimensional data" synonymously with "OLAP data."

Object Linking and Embedding Database (OLE DB)

Object Linking and Embedding Database (OLE DB)

Microsoft's set of Component Object Model (COM) interfaces that provide applications with uniform access to data stored in diverse information sources. OLE DB also provides the ability to implement additional database services.

Object Management Group (OMG)

An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications.

For more information, see <http://www.omg.org/>.

ODBO

Acronym for OLE DB for OLAP.

OLAP

Acronym for online analytical processing.

OLAP Command Processor

Part of the OLAP Query API, an interface that makes it easier to use the API by hiding the complexity of the underlying OLAP Query Model. With this interface, you can create and manipulate complex multidimensional queries with simple commands.

OLAP data provider (ODP)

Provides data in multidimensional views and metadata compatible with the OLAP Metadata Model.

OLAP Metadata API

A set of interfaces provided by the BI Java SDK for accessing the metadata of an OLAP data source. Generated via JMI from the SDK's OLAP Metadata Model.

OLAP Metadata Model

OLAP Metadata Model

A model provided by the BI Java SDK that exposes business data in a multidimensional format which specifically supports data analysis. Based on the CWM OLAP package.

OLAP Query API

A set of interfaces provided by the BI Java SDK that let you define queries against an OLAP server. The API is generated via JMI from the OLAP Query Model, based on metadata provided by the OLAP Metadata Model, and includes the simplified OLAP Command Processor.

OLAP Query Model

An abstraction layer, or model, in the BI Java SDK designed for formulating OLAP queries independently of data source-specific query APIs. The model is based on the CWM-compliant metadata provided by the OLAP Metadata Model.

OLAP Table Model

A companion to the BI Java SDK's ResultSet API that facilitates the rendering of a multidimensional dataset into a two-dimensional matrix.

OLE DB

Acronym for Object Linking and Embedding Database.

OLE DB for OLAP (ODBO)

Microsoft's set of objects and interfaces that extend the ability of OLE DB to provide access to multidimensional data sources on the Windows platform. Providers of OLAP data can implement the interfaces described with OLE DB for OLAP to allow all OLAP clients to access their data.

OLTP

Acronym for online transactional processing.

OMG

OMG

Acronym for Object Management Group.

online analytical processing (OLAP)

A system of organizing data in a multidimensional model that is suitable for decision support. OLAP is the analytical counterpart of OLTP, or online transactional processing. SAP's BI is an OLAP system.

online transactional processing (OLTP)

A system of organizing data in a highly normalized relational model that is suitable for transactional support with frequent update operations. SAP's R/3 is an OLTP system.

Portal Connection Framework

Part of the SAP NetWeaver Portal, provides a set of APIs which extend the standard JCA interfaces and are used to build Portal-compliant connectors. The BI Java Connectors are compliant with the Portal Connection Framework.

query API

Sets of interfaces provided by the BI Java SDK for creating queries against data sources. They are generated via JMI from the SDK's query models, providing methods to create and execute complex OLAP or relational queries based on the metadata in the SDK's CWM-based metadata models.

The SDK provides two query APIs:

- OLAP Query API, for defining queries against an OLAP server
- Relational Query API, for defining queries upon relational data sources

query model

An object-oriented abstraction layer, or model, upon which to formulate queries on a variety of resources without being tied to a specific protocol or query language, such as MDX or SQL. The query models are the basis of their respective query APIs. Two query models are provided by the SDK:

- OLAP Query Model
- Relational Query Model

RDBMS

RDBMS

Acronym for relational database management system.

Relational Command Processor

Part of the SDK's Relational Query API, an interface that makes it easier to use the API by hiding the complexity of the underlying Relational Query Model. With this interface, you can create and manipulate complex relational queries with simple commands.

relational data

Data stored in tables, and hence often also referred to as tabular data. Synonymous with tabular data.

relational data provider (RDP)

Provides data in relational, or tabular, views and metadata compatible with the Relational Metadata Model.

relational database

A repository for typically large amounts of information, structured in accordance with the relational model, in tables with columns. A relational database is created and administered by a relational database management system (RDBMS).

relational database management system (RDBMS)

A system that allows you to create and administer relational databases. With an RDBMS, you define storage structures for data and mechanisms for its manipulation and retrieval (typically via SQL). An RDBMS must also provide a system for safeguarding in case of events such as system crashes and unauthorized access.

Relational Metadata API

A set of interfaces provided by the BI Java SDK for accessing the metadata of a relational data source. Generated via JMI from the SDK's Relational Metadata Model.

Relational Metadata Model

Relational Metadata Model

A model provided by the BI Java SDK that describes data accessible through a relational interface such as JDBC. Based on the CWM Relational Package.

Relational Query API

A set of interfaces provided by the BI Java SDK that let you define queries against a relational data source. The API is generated via JMI from the Relational Query Model, based on metadata provided by the Relational Metadata Model, and includes the simplified Relational Command Processor.

Relational Query Model

An abstraction layer, or model, in the BI Java SDK designed for formulating relational queries independently of data source-specific query APIs. The model is based on the CWM Expressions package and the CWM-compliant metadata provided by the Relational Metadata Model.

resource adapter

As defined by the JCA specification, a system-level software driver component used to connect to an EIS. The BI Java SDK and UD Connect use resource adapters called BI Java Connectors.

resource adapter archive (RAR)

Complete resource adapter modules, which as defined by the JCA specification consist of the required Java classes, documentation, native libraries, and deployment descriptors necessary to distribute a given resource adapter (connector). The BI Java Connectors are distributed in RAR files.

resource adapter module

A complete resource adapter which, as specified by the JCA, is represented physically by a RAR file.

ResultSet API

A set of interfaces that provide applications created with the BI Java SDK with access to the results of a query. The ResultSet API provides access to a relational result set from a relational data source, and an OLAP result set from an OLAP data source.

row

row

A set of fields within a table that contains the data for one specific entry in the table. Each row in a given table has the same structure, predefined for a particular table.

SAP Query

A component of SAP's Web Application Server that allows you to create custom reports without any ABAP programming knowledge. The BI SAP Query Connector uses SAP Query to allow applications created with the BI Java SDK to access data from these SAP operational applications.

For more information about SAP Query, refer to the SAP Help Portal documentation:

http://help.sap.com/saphelp_nw04/helpdata/en/d2/CB3EFB455611D189710000E8322D00/frameset.htm

SDK

Software development kit. As applied throughout this documentation, the term "SDK" refers specifically to the BI Java SDK, unless otherwise specified.

Service Provider Interface (SPI)

Defined by the JCA, a standard interface for integrating the transaction, security, and connection management facilities of an application server with those of a transactional resource manager.

System Landscape

A service of the SAP NetWeaver Portal that provides the functionality to persist connection information.

table

A set of rows, also known as a relation. The table is the central object of the relational model.

tabular data

Synonym for relational data.

UD Connect (UDC)

UD Connect (UDC)

Component of SAP BI that, together with the SAP Web AS J2EE server, provides connectivity to virtually all relational and multidimensional data sources. UDC uses the BI Java Connectors as resource adapters for establishing connections to data sources. The data can either be loaded into BI, or accessed directly via a RemoteCube.

Read about UD Connect on the SAP Help Portal at:

http://help.sap.com/saphelp_nw04/helpdata/en/78/EF1441A509064ABEE6FFD6F38278FD/frameset.htm

XML Metadata Interchange (XMI)

OMG's XML-based standard for interchanging metadata between UML-based modeling tools and MOF-based metadata repositories in distributed, heterogeneous development environments. The exchange takes place in the form of data streams or files, essentially formally mapping MOF to XML.

XMI, together with UML and MOF, forms the core of the OMG's metadata repository architecture and allows developers of distributed systems to share object models and other metadata over the Internet. The SDK supports XMI to interchange and persist metadata objects in a platform independent manner.

For more information, see <http://www.omg.org/technology/documents/formal/xmi.htm>.

XML for Analysis (XMLA)

An XML-messaging-based protocol specified by Microsoft for exchanging analytical data between client applications and servers (for example, OLAP providers) using HTTP and SOAP as a service on the Web. XMLA advances OLE DB concepts such as schema rowsets and MDX by providing standardized universal data access without the need to deploy client components that expose COM interfaces. XML for analysis is not restricted to any particular platform, application, or development language.

For more information, see <http://www.xmla.org/>.

Index

Page numbers in **bold** indicate glossary pages.

Page numbers in *italics* indicate diagrams.

A

ABAP, 40

ActiveX Data Objects, 39, 108, **D-1**

ActiveX Data Objects Multidimensional, 39, **D-1**

adapter classes, 87

addMember, 102

ADO. *See ActiveX Data Objects*

ADO MD. *See ActiveX Data Objects Multidimensional*

aggregate, 60

ALL level, 49

ALL member, 49

APIs

BI Java SDK, 2

APIs of the BI Java SDK, 2-3

attribute, 52

Attribute, 72, 83, 93

attributes

and CWM, 52

metadata mapping, 54

axis cursors, 108

AxisDimension, 72

B

Backus-Naur Form, 96, 98

BEx query mapping, 56

BEx result set, 14

BI Java Connector, **D-1**

BI Java Connectors, 2, 5, 7, 32, 34, 37-40, A-5-A-7

BI JDBC Connector, 39

BI ODBO Connector, 39

BI SAP Query Connector, 40

BI XMLA Connector, 40

language property, 125

testing, A-7

BI Java SDK, **D-3**

APIs, 2

open standards, 5-10

overview, 1

BI JDBC Connector, 4, 7, 39, **D-2**

BI ODBO Connector, 4, 39, 53, **D-2**

BI SAP Query Connector, 4, 40, **D-2**

BI XMLA Connector

note on using in non-managed environment, A-3

BI XMLA Connector, 5, 40, 53

BI XMLA Connector, **D-2**

BIDataSetTableModel, 116

BIException, 125

BIResourceException, 125

BIRuntimeException, 125

Business Information Warehouse, 1, 5, 40

supported version, 11

business intelligence, 1

C

- calculated members, 104
- cartesian product, 59
- catalogs, 58
- CCI. *See* Common Client Interface
- cell cursor, 108
- characteristic, D-7
- classpath configuration, A-2
- column, **D-3**
- columns, 58
- command processor, **D-3**
- command processors, 64
- Common Client Interface, 32, 33, **D-3**
- Common Warehouse Metamodel, 5, 8, 42-43, **D-4**
 - Javadocs, 45
 - supported version, 11
- complex value selection, 104
- composite design pattern, 77
- connection architecture, 32-40
- connection interfaces, 2, 3
- connection management, 32
- ConnectionSpec, 36, 37
- connectivity flow, 38
- connector, **D-4**
- Connector Gateway, **D-4**
- CORBA, 9
- cube, 47, 71, **D-4**
 - mapping notes, 55
- cubes
 - metadata mapping, 53
- cursor model, 109
- cursors, 108
- CWM. *See* Common Warehouse Metamodel
 - additional resources, C-2
 - and attributes, 52
 - and measures, 52

- and members, 52

- Javadocs, 45

- CWM Expressions, 95

- CWM Javadocs, 45

- CWM OLAP model, 52

- CWM OLAP Model, 51

- CWM Relational, 60

- CWM Relational Package, 61

D

- data set, 108

- decomposed, 114

- flattened, 112

- sample, 112

- data set decomposition, 112, 114

- data set flattening, 112

- DataValue, 83

- default members, 48

- deployment descriptor, **D-4**

- dimension, **D-4**

- OLAP vs. SAP BI, D-5

- Dimension, 83

- dimension attributes, 103

- and CWM, 52

- dimensions, 47

- metadata mapping, 53

- documentation, A-4

- resolving links, A-4

- drill down, 50, 104

- drill up, 50, 104

E

- EIS. *See* Enterprise Information System

- Enterprise Information System, 7, 32, **D-5**

- examples, 4

- configuring your system, B-2

- connection properties, B-3

- connector and data source requirements, B-2

- finding the examples, B-1
- Helpers.java, 41, B-5
- HTML stylesheet, 123
- index, B-3
- OLAP Table Model, 118
 - Helpers.java, 123
- OLAP Tutorial, 14-23, 40
- Olap_1.java, 62, B-4
- Olap_10.java, 104, B-5
- Olap_2.java, 101, B-4
- Olap_3.java, 101, B-4
- Olap_4.java, 103, B-4
- Olap_5.java, 103, B-4
- Olap_6.java, 103, B-4
- Olap_7.java, 103, B-4
- Olap_8.java, 103, B-4
- Olap_9.java, B-4
- Relational Tutorial, 23-30, 40
- Relational_1.java, 63, B-5
- Relational_2.java, 63, B-5
- Relational_3.java, 104
- Relational_3.java, B-5
- Relational_4.java, 104
- Relational_4.java, B-5
- Relational_5.java, 104
- Relational_5.java, B-5
- rendering to file, B-3
- Tutorial_1.java, B-3
- Tutorial_2.java, B-3
- exception framework, 124, 125
- exceptions
 - specifying language, 125
- F**
- Feature, 83
- filter, **D-5**
- filtering, 18, 103

- G**
- generating interfaces with JMI, 43-45
- getCatalog(), 63
- getCube(), 62
- getMemberData(), 62
- getMemberData(List, List), 63
- getMetadata(), 110
- getSchema(), 62, 63
- getTable(), 63
- getTaggedValue(), 62
- glossary, D-1-D-15
- H**
- Helpers.java, 15, 41, 123, B-5
- hierarchies, 48, 49
 - mapping notes, 57
 - member drill operations, 103
 - metadata mapping, 54
 - selecting members, 49
- hierarchy, **D-5**
- How-To Guides, A-4
- howto.html files, 38, A-5, A-7
- HTML stylesheet, 123
- I**
- IBIAddition, 89
- IBIAncessor, 85
- IBIAnyDataValue, 83
- IBIAttributeReference, 87
- IBIAxis, 69, 70, 72
- IBIAxisCursor, 109
- IBIAxisDimension, 69, 72
- IBIAxisDimensions, 71
- IBICalculatedMember, 72, 83, 84
- IBIChildren, 80
- IBIClosingPeriod, 85
- IBICommandProcessor, 65
- IBICompositeMemberSetExpression, 75, 79

- IBICompositeTupleSetExpression, 75, 81
- IBIConcatenation, 89
- IBIConditionBasedFilter, 75
- IBIConnection, 15, 24, 26, 32, 33
- IBICousin, 85
- IBICurrentMember, 83, 84
- IBIDataset, 108
- IBIDataSet, 20, 109
- IBIDescendantsLevel, 80
- IBIDimensionAttributeCursor, 110
- IBIDimensionCursor, 110
- IBIDimensionMembers, 80
- IBIDivision, 89
- IBIDrill, 79, 81
- IBIDrillDirectonType, 90
- IBIFilter, 75
- IBIFirstChild, 85
- IBIFirstSibling, 85
- IBIHierarchyMembers, 80
- IBIIDentifiable, 83
- IBIInputReference, 72, 86, 87
- IBIJoinType, 90
- IBILag, 85
- IBILastChild, 85
- IBILastPeriods, 80
- IBILastSibling, 85
- IBILead, 85
- IBILevelDrill, 79
- IBILevelMembers, 80
- IBILevelType, 90
- IBILiteralReference, 87
- IBIMember, 72, 82, 83, 85
- IBIMemberDrill, 79
- IBIMemberExpression, 83, 84, 85
- IBIMemberList, 80
- IBIMemberReference, 87
- IBIMemberSelection, 79, 80
- IBIMemberSet, 69, 73
- IBIMemberSetExpression, 69, 73, 74, 75, 79
- IBIMultiplication, 89
- IBINextMember, 85
- IBINumericValueFunction, 87
- IBIOlap, 17, 32, 33, 39, 40, **D-5**
- IBIOpeningPeriod, 85
- IBIOperation, 87, 89
- IBIParallelPeriod, 85
- IBIParent, 85
- IBIPeriodsToDate, 80
- IBIPreviousMember, 85
- IBIQuery, 18, 27, 69
- IBIRange, 80
- IBIRankingFilter, 75
- IBIRankType, 90
- IBIRelational, 26, 32, 33, 39, 40, **D-5**
- IBIRelationalOperatorType, 90
- IBISapHierarchyValue, 94
- IBISapHierarchyVariable, 93
- IBISapMemberValueRange, 94
- IBISapMemberVariable, 93
- IBISapNumericValueRange, 94
- IBISapNumericVariable, 93
- IBISapRangeSignType, 90
- IBISapVariable, 93
- IBISapVariableSelectionType, 90
- IBISapVariableValue, 94
- IBISapVariableValueRange, 94
- IBISet, 69
- IBISignInversion, 89
- IBISort, 75
- IBISortDirectionType, 90
- IBISubtraction, 89
- IBITimeSeries, 80

- IBITuple, 81
- IBITupleDrill, 81
- IBITupleList, 73, 81
- IBITupleReference, 87
- IBITupleSelection, 81
- IBITupleSet, 69, 72
- IBITupleSetExpression, 69, 73, 74, 75, 81
- IConnectionSpec, 36
- icons, iii
- INative, 34, **D-5**
- InfoCube, D-4
- InfoObject (characteristic), D-5
- InfoProvider
 - access to the data of an, 55
- InputReference, 87
- installation process, A-1
- Instance, 83, 94
- iViews, A-4
- J**
- J2EE, 6, A-5
- J2EE Connector Architecture, 7, 31, 32, **D-6**
 - supported version, 11
- JARM, A-2
- Java Application Responsetime Management, A-2
- Java Database Connectivity, 1, 9, 39, 58, **D-6**
 - connection URL, 25
 - supported version, 11
- Java Development Kit
 - supported version, 11
- Java Metadata Interface, 5, 8, 43, **D-6**
 - generating interfaces with, 43-45
 - supported version, 11
- java.sql.ResultSet, 107
- Javadocs, 3
 - CWM, 45
- JCA. *See J2EE Connector Architecture*
 - additional resources, C-1
- JCo, 7, **D-6**
- JDBC. *See Java Database Connectivity*
 - additional resources, C-1
- JDBC connection URL, 25
- JDK. *See Java Development Kit*
- JMI. *See Java Metadata Interface*
 - additional resources, C-2
- JMI process, 44
- JMI service, 7, 8, 43, **D-7**
- join types, 76, 91
- K**
- key figure, D-7
- key figures, 48
- L**
- language property, 125
- layout
 - changing, 19
- legacy exceptions, 124
- level, **D-7**
- levels, 48, 49
 - ALL level, 49
 - mapping notes, 57
 - metadata mapping, 54
 - selecting members based on, 49
- logging and tracing, A-2
- M**
- Main Model, 68, 69, 93
- managed application scenario, 35
- managed environment, 34
- mapping OLAP metadata, 51-57
- mapping relational metadata, 60-62
- MDX, 1, 5, 10, 12, **D-7**
 - additional resources, C-2
 - direct execution of a statement, 101
- measure, 52, **D-7**

- measures, 48
 - and CWM, 52
 - mapping notes, 56
 - metadata mapping, 54
 - member, 52, **D-7**
 - ALL member, 49
 - default member, 49
 - Member, 82, 83
 - member selection, 49
 - member set expressions, 78
 - MemberExpression, 85
 - members, 48
 - ALL members, 49
 - and CWM, 52
 - metadata mapping, 55
 - selecting, 20, 49
 - MemberSelection, 80
 - MemberSetExpression, 75, 79
 - Meta Object Facility, 9, 44, **D-7**
 - supported version, 11
 - metadata
 - accessing OLAP, 62
 - accessing relational, 63
 - retrieving multidimensional, 18
 - retrieving relational, 27
 - metadata access
 - in OLAP result sets, 110
 - in relational result sets, 115
 - metadata API, **D-8**
 - metadata APIs, 42, 45
 - metadata instances, 45
 - metadata mapping, 53
 - metadata model, 42, **D-8**
 - metadata models, 45
 - metadata repository, **D-8**
 - Metamodel Repository, 7, 43, **D-8**
 - MMR. *See Metamodel Repository*
 - MOF. *See Meta Object Facility*
 - additional resources, C-2
 - moveDimensionToColumns, 102
 - moveDimensionToRows, 102
 - multidimensional data, **D-8**
 - multidimensional result set, 67
 - multidimensional result sets, 111
- N**
- NetWeaver, 6
 - supported version, 11
 - NetWeaver Portal, 7, 34, A-4
 - non-managed environment, 35
 - NumericValueFunction, 88
 - NumericValueFunctions, 88
- O**
- Object, 83
 - Object Linking and Embedding Database, **D-9**
 - Object Management Group, 8, 42, **D-9**
 - ObjectFinder, 18, 62, 63
 - ODBO. *See OLE DB for OLAP*
 - ODP. *See OLAP data provider*
 - OLAP. *See Online Analytical Processing*
 - OLAP BAPI, 53
 - OLAP Command Processor, 19, 64, 66, **D-9**
 - OLAP cube, 47
 - OLAP data provider, **D-9**
 - OLAP metadata
 - mapping of, 51-57
 - OLAP Metadata API, 2, 3, **D-9**
 - OLAP Metadata Model, 8, 46-57, 46, 66, **D-10**
 - OLAP queries
 - examples of, 101
 - OLAP Query API, 2, 3, 10, **D-10**
 - OLAP Query Model, 5, 65-92, **D-10**
 - Main Model, 68

- OLAP Result Sets, 108
- OLAP systems, 46–50, 46
- OLAP Table Model, 21, 116–17, **D–10**
 - algorithm, 117
- OLAP Tutorial, 14–23
- Olap_1.java, 62, B–4
- Olap_10.java, 104, B–5
- Olap_2.java, 101, B–4
- Olap_3.java, 101, B–4
- Olap_4.java, 103, B–4
- Olap_5.java, 103, B–4
- Olap_6.java, 103, B–4
- Olap_7.java, 103, B–4
- Olap_8.java, 103, B–4
- Olap_9.java, B–4
- OLE DB. *See Object Linking and Embedding Database*
- OLE DB for OLAP, 1, 4, 10, 39, 53, 108, **D–10**
- OLTP. *See Online Transactional Processing*
- OMG. *See Object Management Group*
- Online Analytical Processing, 1, **D–11**
- Online Transactional Processing, **D–11**
- Open Analysis Interfaces, 1
 - OLAP BAPI, 1
 - OLE DB for OLAP, 1
 - XML for Analysis, 1
- Open Database Connectivity, 58
- open standards, 5–10
- Operation, 89
- Operations, 89
- operators, 59
- P**
- pivot, 50
- pivoting layout, 101
- Portal Connection Framework, 33, 34, **D–11**
- project, 59
- properties, 48
- Q**
- query
 - creating multidimensional, 18
 - creating relational, 27
 - executing multidimensional, 20
 - executing relational, 28
 - new query operation, 102
- query (SAP BI), D–4
- query API, **D–11**
- query model, **D–11**
- R**
- R/3, 7
- RAR. *See resource adapter archive*
- RAR files, 38
- RDBMS, 57, *See relational database management system*
- RDP. *See relational data provider*
- references
 - books, C–3
- relational
 - complex relational query, 104
 - operators, 59
 - simple relational query, 104
- relational algebra, 59
- Relational Command Processor, 27, 64, 95, **D–12**
- relational data, **D–12**
- relational data provider, **D–12**
- relational database, **D–12**
- relational database management system, **D–12**
- relational databases, 57–58
- relational metadata
 - mapping of, 60–62
- Relational Metadata API, 2, 3, **D–12**
- Relational Metadata Model, 8, 57–62, 57, **D–13**
- relational operators

- aggregate, 60
- cartesian product, 59
- project, 59
- rename, 59
- select, 59
- set difference, 59
- sort, 60
- union, 59
- relational queries
 - examples of, 104
- Relational Query API, 2, 3, **D-13**
- Relational Query Model, 5, 95–101, **D-13**
 - SQL subset, 96
- relational result sets, 115
- Relational Tutorial, 23–30
- Relational_1.java, 63, B–5
- Relational_2.java, 63, B–5
- Relational_3.java, 104, B–5
- Relational_4.java, 104
- Relational_4.java, B–5
- Relational_5.java, 104
- Relational_5.java, B–5
- release notes, A–1
- rename, 59
- resource adapter, 32, 34, 37, **D-13**
- resource adapter archive, **D-13**
- resource adapter module, **D-13**
- result set
 - rendering, 20, 29
- ResultSet, 28
- ResultSet API, 2, 3, 9, 106–16, **D-13**
 - cursor model, 109
 - OLAP data access, 111
 - OLAP metadata access, 110
 - OLAP navigational aspects, 110
 - relational data access, 116
 - relational metadata access, 115
 - relational navigational aspects, 115
- row, **D-14**
- rows, 58
- S**
- sample data set, 112
- SAP Exception Framework, 124
- SAP Query, 2, 40, 98, **D-14**
 - additional resources, C–1
- SAP Sales DemoCube: Overview, 55
- SAP variables, 92, 104
 - additional resources, C–1
 - complex value selection, 94
 - complex value selection, 104
 - metadata mapping, 55
 - variable values, 94
- schema
 - mapping notes, 55
- schemas, 58
 - metadata mapping, 53
- SDK, **D-14**
- select, 59
- selecting members, 49
- SELECT-OPTIONS, 99
- Service Provider Interface, 32, 34, **D-14**
- set difference, 59
- slicer, 19, 70
- Slot, 69, 83, 93, 94
- sort, 60
- sorting
 - sort by dimension attribute, 103
 - sort by measure value, 103
- SPI. *See Service Provider Interface*
- SQL, 1, 8, 9, 12, 58, *See also Structured Query Language*
 - direct execution of statement, 104

StructuralFeature, 83, 93
structure element, D-7
Structured Query Language, 58, *See also* SQL
System Landscape, **D-14**
system requirements, A-1

T

table, 58, **D-14**
tabular data, **D-14**
testing
 testing the BI Java Connectors, A-7
top N query, 50
tree-form representation, 100
tuple, 47, 58
tuples, 58
TupleSetExpression, 75, 81
tutorial
 OLAP, 14-23
 relational, 23-30
Tutorial 1 Result Set, 23
Tutorial 2 Sample Result Set, 30
Tutorial_1.java, 14-23, 40, B-3
Tutorial_2.java, 23-30, 40, B-3
types, 89
Types, 90
typographical conventions, iii

U

UD Connect, **D-15**
 additional resources, C-1

UDC. *See* *UD Connect*

UML metadata, 45
Unified Modeling Language, 9, 44
union, 59

V

Variable Values, 94
variables, 92
Variables, 93

W

Web Application Server, 6, 34, 40
Web Dynpro applications, A-5

X

XMI. *See* *XML Metadata Interchange*
 additional resources, C-2
XML for Analysis, 2, 10, 40, 53, **D-15**
XML Metadata Interchange, 9, **D-15**
 supported version, 11

XMLA

 BI XMLA Connector
 note on using in non-managed environment, A-3

XMLA

 additional resources, C-1

XMLA. *See* *XML for Analysis*

XMLA provider, 16

Z

zoom in, 104
zoom out, 104