Oracle® Fusion Middleware

Knowledge Module Developer's Guide for Oracle Data Integrator

11*g* Release 1 (11.1.1)

E12645-04

February 2013

ORACLE

Oracle Fusion Middleware Knowledge Module Developer's Guide for Oracle Data Integrator, 11*g* Release 1 (11.1.1)

E12645-04

Primary Author: Laura Hofman Miquel

Contents

Preface ix

1 Introduction to Knowledge Modules

1.1 What is a Knowledge Module? 1-1
1.2 Reverse-Engineering Knowledge Modules (RKM) 1-2
1.3 Check Knowledge Modules (CKM) 1-3
1.4 Loading Knowledge Modules (LKM) 1-4
1.5 Integration Knowledge Modules (IKM) 1-5
1.6 Journalizing Knowledge Modules (JKM) 1-6
1.7 Service Knowledge Modules (SKM) 1-7
1.8 Guidelines for Knowledge Module Developers 1-7

2 Introduction to the Substitution API

2.1 Introduction to the Substitution API 2-1
2.2 Using Substitution Methods 2-2
2.2.1 Generic Syntax 2-2
2.2.2 Specific Syntax for CKM 2-3
2.2.3 Using Flexfields 2-3
2.3 Using Substitution Methods in Actions 2-3
2.3.1 Action Lines Code 2-4
2.3.2 Action Calls Method 2-4
2.4 Working with Object Names 2-5
2.5 Working with Lists of Tables, Columns and Expressions 2-6
2.5.1 Using getTargetColList to create a table 2-7
2.5.2 Using getColList in an Insert values statement 2-8
2.5.3 Using getSrcTableList 2-8
2.6 Generating the Source Select Statement 2-9
2.7 Working with Datasets 2-11
2.8 Obtaining Other Information with the API 2-11
2.9 Advanced Techniques for Code Generation 2-12

3 Reverse-Engineering Strategies

3.1 Customized Reverse-Engineering Process 3-1
3.1.1 SNP_REV tables 3-1
3.1.2 Customized Reverse-Engineering Strategy 3-1

3.2 Case Studies ... 3-2
3.2.1 RKM Oracle ... 3-2
3.2.1.1 Reset SNP_REV Tables ... 3-2
3.2.1.2 Get Tables ... 3-2
3.2.1.3 Get views, partitions, columns, FK, Keys and other Oracle Metadata ... 3-3
3.2.1.4 Set Metadata ... 3-3

4 Data Integrity Strategies

4.1 Data Integrity Check Process ... 4-1
4.1.1 Check Knowledge Module Overview ... 4-1
4.1.2 Error Tables Structures ... 4-2
4.1.2.1 Error Table Structure ... 4-2
4.1.2.2 Summary Table Structure ... 4-3
4.2 Case Studies ... 4-4
4.2.1 Oracle CKM ... 4-4
4.2.1.1 Drop Check Table ... 4-4
4.2.1.2 Create Check Table ... 4-4
4.2.1.3 Create Error Table ... 4-5
4.2.1.4 Insert PK Errors ... 4-5
4.2.1.5 Delete Errors from Controlled Table ... 4-6
4.2.2 Dynamically Create Non-Existing References ... 4-6
4.2.2.1 Use Case ... 4-6
4.2.2.2 Discussion ... 4-7
4.2.2.3 Implementation Details ... 4-8

5 Loading Strategies

5.1 Loading Process ... 5-1
5.1.1 Loading Process Overview ... 5-1
5.1.2 Loading Table Structure ... 5-1
5.1.3 Loading Method ... 5-2
5.1.3.1 Loading Using the Agent ... 5-2
5.1.3.2 Loading File Using Loaders ... 5-2
5.1.3.3 Loading Using Unload/Load ... 5-3
5.1.3.4 Loading Using RDBMS-Specific Strategies ... 5-3
5.2 Case Studies ... 5-3
5.2.1 LKM SQL to SQL ... 5-4
5.2.1.1 Drop Work Table ... 5-4
5.2.1.2 Create Work Table ... 5-4
5.2.1.3 Load Data ... 5-4
5.2.1.4 Drop Work Table ... 5-5

6 Integration Strategies

6.1 Integration Process ... 6-1
6.1.1 Integration Process Overview ... 6-1
6.1.2 Integration Strategies ... 6-2
6.1.2.1 Strategies with Staging Area on the Target ... 6-2

6.1.2.2 Strategies with the Staging Area Different from the Target ... 6-7
6.2 Case Studies ... 6-8
6.2.1 Simple Replace or Append ... 6-8
6.2.1.1 Delete Target Table ... 6-9
6.2.1.2 Insert New Rows ... 6-9
6.2.2 Backup the Target Table Before Loading ... 6-10
6.2.2.1 Drop Backup Table ... 6-10
6.2.2.2 Create Backup Table ... 6-10
6.2.3 Tracking Records for Regulatory Compliance ... 6-10
6.2.3.1 Load Tracking Records ... 6-11

A Substitution API Reference

A.1 Substitution Methods List ... A-1
A.1.1 Global Methods ... A-1
A.1.2 Journalizing Knowledge Modules ... A-2
A.1.3 Loading Knowledge Modules ... A-2
A.1.4 Check Knowledge Modules ... A-3
A.1.5 Integration Knowledge Modules ... A-3
A.1.6 Reverse-Engineering Knowledge Modules ... A-4
A.1.7 Service Knowledge Modules ... A-4
A.1.8 Actions ... A-5
A.2 Substitution Methods Reference ... A-5
A.2.1 getAK() Method ... A-5
A.2.2 getAKColList() Method ... A-6
A.2.3 getAllTargetColList() Method ... A-9
A.2.4 getCatalogName() Method ... A-9
A.2.5 getCatalogNameDefaultPSchema() Method ... A-10
A.2.6 getCK() Method ... A-11
A.2.7 getColDefaultValue() Method ... A-12
A.2.8 getColList() Method ... A-13
A.2.9 getColumn() Method ... A-18
A.2.10 getContext() Method ... A-20
A.2.11 getDataSet() Method ... A-21
A.2.12 getDataSetCount() Method ... A-22
A.2.13 getDataType() Method ... A-23
A.2.14 getFilter() Method ... A-24
A.2.15 getFilterList() Method ... A-24
A.2.16 getFK() Method ... A-26
A.2.17 getFKColList() Method ... A-27
A.2.18 getFlexFieldValue() Method ... A-30
A.2.19 getFrom() Method ... A-31
A.2.20 getGrpBy() Method ... A-32
A.2.21 getGrpByList() Method ... A-33
A.2.22 getHaving() Method ... A-34
A.2.23 getHavingList() Method ... A-35
A.2.24 getIndex() Method ... A-37
A.2.25 getIndexColList() Method ... A-37

A.2.26 getInfo() Method A-39
A.2.27 getJDBCConnection() Method A-43
A.2.28 getJDBCConnectionFromLSchema() Method A-43
A.2.29 getJoin() Method A-44
A.2.30 getJoinList() Method A-45
A.2.31 getJrnFilter() Method A-46
A.2.32 getJrnInfo() Method A-47
A.2.33 getLoadPlanInstance() Method A-48
A.2.34 getModel() Method A-49
A.2.35 getNbInsert(), getNbUpdate(), getNbDelete(), getNbErrors() and getNbRows() Methods A-50
A.2.36 getNewColComment() Method A-51
A.2.37 getNewTableComment() Method A-51
A.2.38 getNotNullCol() Method A-51
A.2.39 getObjectName() Method A-52
A.2.40 getObjectNameDefaultPSchema() Method A-55
A.2.41 getOption() Method A-56
A.2.42 getPackage() Method A-56
A.2.43 getParentLoadPlanStepInstance() Method A-57
A.2.44 getPK() Method A-57
A.2.45 getPKColList() Method A-58
A.2.46 getPop() Method A-61
A.2.47 getPrevStepLog() Method A-62
A.2.48 getQuotedString() Method A-64
A.2.49 getSchemaName() Method A-64
A.2.50 getSchemaNameDefaultPSchema() Method A-65
A.2.51 getSession() Method A-67
A.2.52 getSessionVarList() Method A-67
A.2.53 getSrcColList() Method A-68
A.2.54 getSrcTablesList() Method A-70
A.2.55 getStep() Method A-73
A.2.56 getSubscriberList() Method A-74
A.2.57 getSysDate() Method A-75
A.2.58 getTable() Method A-76
A.2.59 getTargetColList() Method A-78
A.2.60 getTableName() Method A-82
A.2.61 getTargetTable() Method A-82
A.2.62 getTemporaryIndex() Method A-84
A.2.63 getTemporaryIndexColList() Method A-85
A.2.64 getUser() Method A-86
A.2.65 hasPK() Method A-87
A.2.66 isColAttrChanged() Method A-87
A.2.67 nextAK() Method A-88
A.2.68 nextCond() Method A-89
A.2.69 nextFK() Method A-89
A.2.70 setNbInsert(), setNbUpdate(), setNbDelete(), setNbErrors() and setNbRows() Methods A-90
A.2.71 setTableName() Method A-91

A.2.72 setTaskName() Method A-91

B SNP_REV Tables Reference

B.1 SNP_REV_SUB_MODEL B-1
B.2 SNP_REV_TABLE B-1
B.3 SNP_REV_COL B-3
B.4 SNP_REV_KEY B-4
B.5 SNP_REV_KEY_COL B-4
B.6 SNP_REV_JOIN B-4
B.7 SNP_REV_JOIN_COL B-5
B.8 SNP_REV_COND B-6

Preface

This manual describes how to develop your own Knowledge Modules for Oracle Data Integrator.

This preface contains the following topics:.

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

This document is intended for developers who want to make advanced use of Oracle Data Integrator and customize Knowledge Modules for their integration processes.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following Oracle resources:

- *Oracle Fusion Middleware Getting Started with Oracle Data Integrator*
- *Oracle Fusion Middleware Installation Guide for Oracle Data Integrator*
- *Oracle Fusion Middleware Upgrade Guide for Oracle Data Integrator*
- *Oracle Fusion Middleware Developer's Guide for Oracle Data Integrator*
- *Oracle Fusion Middleware Connectivity and Modules Guide for Oracle Data Integrator*
- *Oracle Fusion Middleware Application Adapters Guide for Oracle Data Integrator*

- *Oracle Data Integrator 11g Online Help*
- *Oracle Data Integrator 11g Release Notes, included with your Oracle Data Integrator 11g installation and on Oracle Technology Network*

Conventions

The following text conventions are used in this document:

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

1

Introduction to Knowledge Modules

This chapter provides an introduction to Knowledge Modules (KM). It explains what is a knowledge module, and describes the different types of KMs.

This chapter includes the following sections:

- Section 1.1, "What is a Knowledge Module?"
- Section 1.2, "Reverse-Engineering Knowledge Modules (RKM)"
- Section 1.3, "Check Knowledge Modules (CKM)"
- Section 1.4, "Loading Knowledge Modules (LKM)"
- Section 1.5, "Integration Knowledge Modules (IKM)"
- Section 1.6, "Journalizing Knowledge Modules (JKM)"
- Section 1.7, "Service Knowledge Modules (SKM)"
- Section 1.8, "Guidelines for Knowledge Module Developers"

1.1 What is a Knowledge Module?

Knowledge Modules (KMs) are code templates. Each KM is dedicated to an individual task in the overall data integration process. The code in the KMs appears in nearly the form that it will be executed except that it includes Oracle Data Integrator (ODI) substitution methods enabling it to be used generically by many different integration jobs. The code that is generated and executed is derived from the declarative rules and metadata defined in the ODI Designer module.

- A KM will be reused across several interfaces or models. To modify the behavior of hundreds of jobs using hand-coded scripts and procedures, developers would need to modify each script or procedure. In contrast, the benefit of Knowledge Modules is that you make a change once and it is instantly propagated to hundreds of transformations. KMs are based on logical tasks that will be performed. They don't contain references to physical objects (datastores, columns, physical paths, etc.)
- KMs can be analyzed for impact analysis.
- KMs can't be executed standalone. They require metadata from interfaces, datastores and models.

KMs fall into 6 different categories as summarized in the table below:

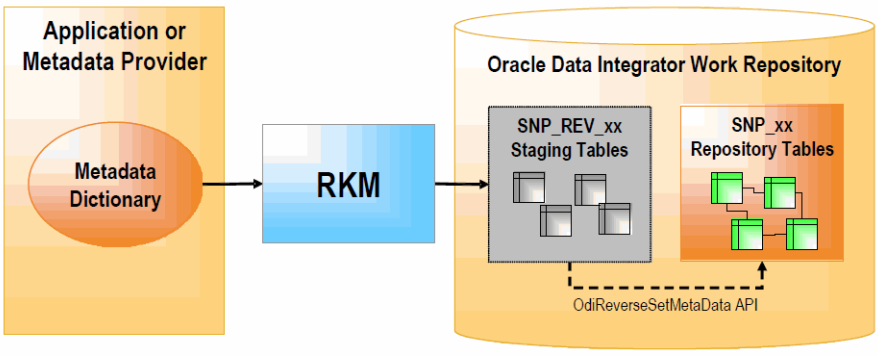
Knowledge Module	Description	Usage
Reverse-engineering KM	Retrieves metadata to the Oracle Data Integrator work repository	Used in models to perform a customized reverse-engineering
Check KM	Checks consistency of data against constraints	▪ Used in models, sub models and datastores for data integrity audit ▪ Used in interfaces for flow control or static control
Loading KM	Loads heterogeneous data to a staging area	Used in interfaces with heterogeneous sources
Integration KM	Integrates data from the staging area to a target	Used in interfaces
Journalizing KM	Creates the Change Data Capture framework objects in the source staging area	Used in models, sub models and datastores to create, start and stop journals and to register subscribers.
Service KM	Generates data manipulation web services	Used in models and datastores

The following sections describe each type of Knowledge Module.

1.2 Reverse-Engineering Knowledge Modules (RKM)

The RKM role is to perform customized reverse engineering for a model. The RKM is in charge of connecting to the application or metadata provider then transforming and writing the resulting metadata into Oracle Data Integrator's repository. The metadata is written temporarily into the SNP_REV_xx tables. The RKM then calls the Oracle Data Integrator API to read from these tables and write to Oracle Data Integrator's metadata tables of the work repository in incremental update mode. This is illustrated below:

Figure 1–1 Reverse-engineering Knowledge Modules



A typical RKM follows these steps:

1. Cleans up the SNP_REV_xx tables from previous executions using the OdiReverseResetTable tool.
2. Retrieves sub models, datastores, columns, unique keys, foreign keys, conditions from the metadata provider to SNP_REV_SUB_MODEL, SNP_REV_TABLE, SNP_

REV_COL, SNP_REV_KEY, SNP_REV_KEY_COL, SNP_REV_JOIN, SNP_REV_JOIN_COL, SNP_REV_COND tables.

3. Updates the model in the work repository by calling the OdiReverseSetMetaData tool.

1.3 Check Knowledge Modules (CKM)

The CKM is in charge of checking that records of a data set are consistent with defined constraints. The CKM is used to maintain data integrity and participates in the overall data quality initiative. The CKM can be used in 2 ways:

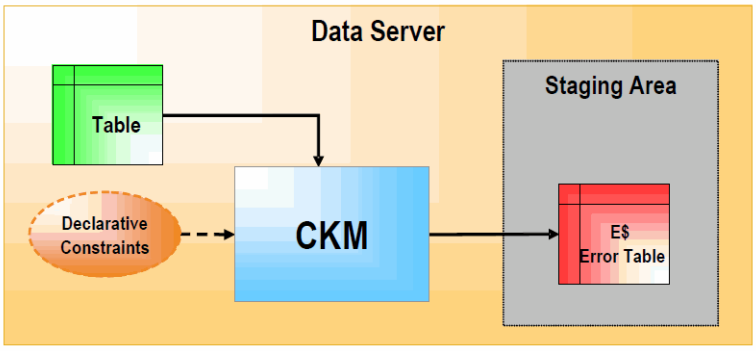
- To check the consistency of existing data. This can be done on any datastore or within interfaces, by setting the STATIC_CONTROL option to "Yes". In the first case, the data checked is the data currently in the datastore. In the second case, data in the target datastore is checked after it is loaded.
- To check consistency of the incoming data before loading the records to a target datastore. This is done by using the FLOW_CONTROL option. In this case, the CKM simulates the constraints of the target datastore on the resulting flow prior to writing to the target.

In summary: the CKM can check either an existing table or the temporary "I$" table created by an IKM.

The CKM accepts a set of constraints and the name of the table to check. It creates an "E$" error table which it writes all the rejected records to. The CKM can also remove the erroneous records from the checked result set.

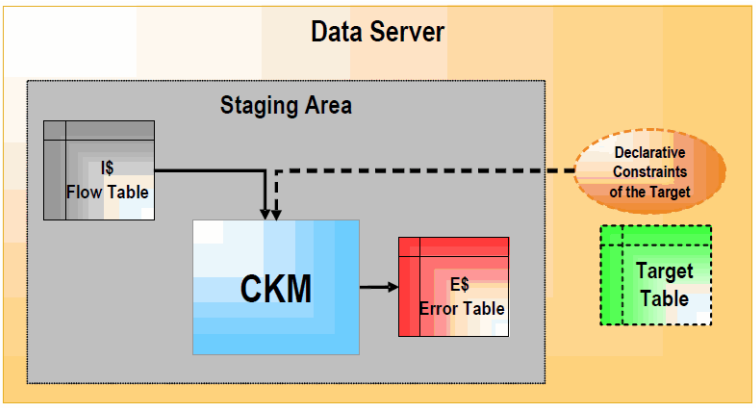
The following figures show how a CKM operates in both STATIC_CONTROL and FLOW_CONTROL modes.

Figure 1–2 Check Knowledge Module (STATIC_CONTROL)



In STATIC_CONTROL mode, the CKM reads the constraints of the table and checks them against the data of the table. Records that don't match the constraints are written to the "E$" error table in the staging area.

Figure 1–3 Check Knowledge Module (FLOW_CONTROL)



In FLOW_CONTROL mode, the CKM reads the constraints of the target table of the Interface. It checks these constraints against the data contained in the "I$" flow table of the staging area. Records that violate these constraints are written to the "E$" table of the staging area.

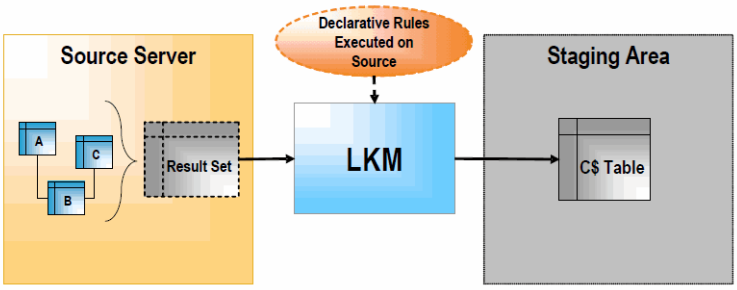
In both cases, a CKM usually performs the following tasks:

1. Create the "E$" error table on the staging area. The error table should contain the same columns as the datastore as well as additional columns to trace error messages, check origin, check date etc.
2. Isolate the erroneous records in the "E$" table for each primary key, alternate key, foreign key, condition, mandatory column that needs to be checked.
3. If required, remove erroneous records from the table that has been checked.

1.4 Loading Knowledge Modules (LKM)

An LKM is in charge of loading source data from a remote server to the staging area. It is used by interfaces when some of the source datastores are not on the same data server as the staging area. The LKM implements the declarative rules that need to be executed on the source server and retrieves a single result set that it stores in a "C$" table in the staging area, as illustrated below.

Figure 1–4 Loading Knowledge Module



1. The LKM creates the "C$" temporary table in the staging area. This table will hold records loaded from the source server.

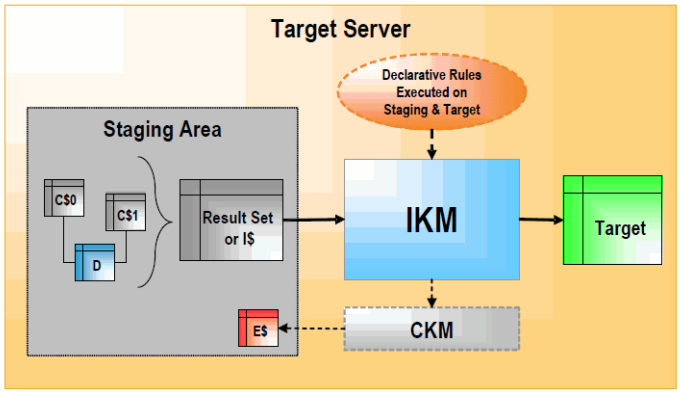
2. The LKM obtains a set of pre-transformed records from the source server by executing the appropriate transformations on the source. Usually, this is done by a single SQL SELECT query when the source server is an RDBMS. When the source doesn't have SQL capacities (such as flat files or applications), the LKM simply reads the source data with the appropriate method (read file or execute API).
3. The LKM loads the records into the "C$" table of the staging area.

An interface may require several LKMs when it uses datastores from different sources. When all source datastores are on the same data server as the staging area, no LKM is required.

1.5 Integration Knowledge Modules (IKM)

The IKM is in charge of writing the final, transformed data to the target table. Every interface uses a single IKM. When the IKM is started, it assumes that all loading phases for the remote servers have already carried out their tasks. This means that all remote source data sets have been loaded by LKMs into "C$" temporary tables in the staging area, or the source datastores are on the same data server as the staging area. Therefore, the IKM simply needs to execute the "Staging and Target" transformations, joins and filters on the "C$" tables, and tables located on the same data server as the staging area. The resulting set is usually processed by the IKM and written into the "I$" temporary table before loading it to the target. These final transformed records can be written in several ways depending on the IKM selected in your interface. They may be simply appended to the target, or compared for incremental updates or for slowly changing dimensions. There are 2 types of IKMs: those that assume that the staging area is on the same server as the target datastore, and those that can be used when it is not. These are illustrated below:

Figure 1–5 Integration Knowledge Module (Staging Area on Target)



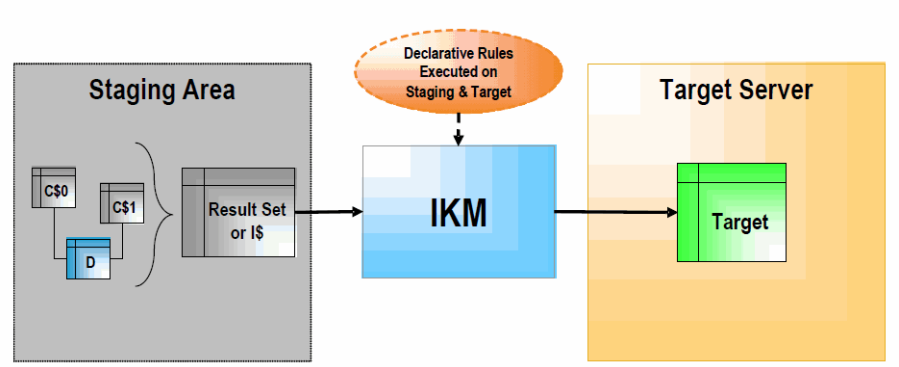
When the staging area is on the target server, the IKM usually follows these steps:

1. The IKM executes a single set-oriented SELECT statement to carry out staging area and target declarative rules on all "C$" tables and local tables (such as D in the figure). This generates a result set.
2. Simple "append" IKMs directly write this result set into the target table. More complex IKMs create an "I$" table to store this result set.
3. If the data flow needs to be checked against target constraints, the IKM calls a CKM to isolate erroneous records and cleanse the "I$" table.

4. The IKM writes records from the "I$" table to the target following the defined strategy (incremental update, slowly changing dimension, etc.).
5. The IKM drops the "I$" temporary table.
6. Optionally, the IKM can call the CKM again to check the consistency of the target datastore.

These types of KMs do not manipulate data outside of the target server. Data processing is set-oriented for maximum efficiency when performing jobs on large volumes.

Figure 1–6 Integration Knowledge Module (Staging Area Different from Target)



When the staging area is different from the target server, as shown in Figure 1–6, the IKM usually follows these steps:

1. The IKM executes a single set-oriented SELECT statement to carry out declarative rules on all "C$" tables and tables located on the staging area (such as D in the figure). This generates a result set.
2. The IKM loads this result set into the target datastore, following the defined strategy (append or incremental update).

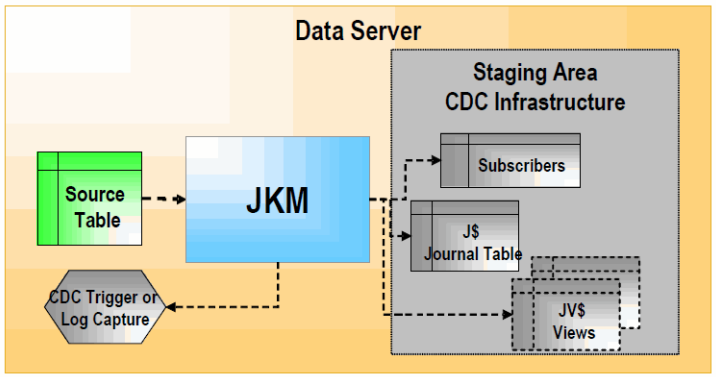
This architecture has certain limitations, such as:

- A CKM cannot be used to perform a data integrity audit on the data being processed.
- Data needs to be extracted from the staging area before being loaded to the target, which may lead to performance issues.

1.6 Journalizing Knowledge Modules (JKM)

JKMs create the infrastructure for Change Data Capture on a model, a sub model or a datastore. JKMs are not used in interfaces, but rather within a model to define how the CDC infrastructure is initialized. This infrastructure is composed of a subscribers table, a table of changes, views on this table and one or more triggers or log capture programs as illustrated below.

Figure 1–7 Journalizing Knowledge Module



1.7 Service Knowledge Modules (SKM)

SKMs are in charge of creating and deploying data manipulation Web Services to your Service Oriented Architecture (SOA) infrastructure. SKMs are set on a Model. They define the different operations to generate for each datastore's web service. Unlike other KMs, SKMs do no generate an executable code but rather the Web Services deployment archive files. SKMs are designed to generate Java code using Oracle Data Integrator's framework for Web Services. The code is then compiled and eventually deployed on the Application Server's containers.

1.8 Guidelines for Knowledge Module Developers

The first guidelines when developing your own KM is to never start from a blank page.

Oracle Data Integrator provides a large number of knowledge modules out-of-the-box. It is recommended that you start by reviewing the existing KMs and start from an existing KM that is close to your use case. Duplicate this KM and customize it by editing the code.

When developing your own KM, keep in mind that it is targeted to a particular stage of the integration process. As a reminder:

- LKMs are designed to load remote source data sets to the staging area into Loading ("C$") tables.
- IKMs apply the source flow from the staging area to the target. They start from the Loading tables, may transform and join them into a single integration table ("I$") table, may call a CKM to perform data quality checks on this integration table, and finally write the flow data to the target
- CKMs check data quality in a datastore or a integration table ("I$") against data quality rules expressed as constraints. The rejected records are stored in the error table ("E$")
- RKMs are in charge of extracting metadata from a metadata provider to the Oracle Data Integrator repository by using the SNP_REV_xx temporary tables.
- JKMs are in charge of creating and managing the Change Data Capture infrastructure.

Be also aware of these common pitfalls:

- Avoid creating too many KMs: A typical project requires less than 5 KMs! Do not confuse KMs and procedures, and do not create one KM for each specific use case. Similar KMs can be merged into a single one and parameterized using options.
- Avoid hard-coded values, including catalog or schema names in KMs: You should instead use the substitution methods getTable(), getTargetTable(), getObjectName(), knowledge module options or others as appropriate.
- Avoid using variables in KMs: You should instead use options or flex fields to gather information from the designer.
- Writing the KM entirely in Jython, Groovy or Java: You should do that if it is the appropriate solution (for example, when sourcing from a technology that only has a Java API). SQL is easier to read, maintain and debug than Java, Groovy or Jython code.
- Using <%if%> statements rather than a check box option to make code generation conditional.

Other common code writing recommendations that apply to KMs:

- The code should be correctly indented.
- The generated code should also be indented in order to be readable.
- SQL keywords such as "select", "insert", etc. should be in lowercase for better readability.

2

Introduction to the Substitution API

This chapter provides an introduction to the Oracle Data Integrator Substitution API using examples.

This chapter includes the following sections:

- Section 2.1, "Introduction to the Substitution API"
- Section 2.2, "Using Substitution Methods"
- Section 2.3, "Using Substitution Methods in Actions"
- Section 2.4, "Working with Object Names"
- Section 2.5, "Working with Lists of Tables, Columns and Expressions"
- Section 2.6, "Generating the Source Select Statement"
- Section 2.7, "Working with Datasets"
- Section 2.8, "Obtaining Other Information with the API"
- Section 2.9, "Advanced Techniques for Code Generation"

> **Note:** The substitution API methods are listed in Appendix A, "Substitution API Reference".

2.1 Introduction to the Substitution API

KMs are written as templates by using the Oracle Data Integrator substitution API. The API methods are java methods that return a string value. They all belong to a single object instance named "odiRef". The same method may return different values depending on the type of KM that invokes it. That's why they are classified by type of KM.

To understand how this API works, the following example illustrates how you would write a create table statement in a KM and what it would generate depending on the datastores it would deal with:

The following code is entered in a KM:

```
CREATE TABLE <%=odiRef.getTable("L", "INT_NAME", "A")%>
(
<%=odiRef.getColList("", "\t[COL_NAME] [DEST_CRE_DT]", ",\n", "", "")%>
)
```

The generated code for the `PRODUCT` table is:

```
CREATE TABLE db_staging.I$_PRODUCT
```

```
(
PRODUCT_ID numeric(10),
PRODUCT_NAME varchar(250),
FAMILY_ID numeric(4),
SKU varchar(13),
LAST_DATE timestamp
)
```

The generated code for the `CUSTOMER` table is:

```
CREATE TABLE db_staging.I$_CUSTOMER
(
CUST_ID numeric(10),
CUST_NAME varchar(250),
ADDRESS varchar(250),
CITY varchar(50),
ZIP_CODE varchar(12),
COUNTRY_ID varchar(3)
)
```

As you can see, once executed with appropriate metadata, the KM has generated a different code for the product and customer tables.

The following topics cover some of the main substitution APIs and their use within KMs. Note that for better readability the tags "<%" and "%>" as well as the "odiRef" object reference are omitted in the examples.

2.2 Using Substitution Methods

The methods that are accessible from the Knowledge Modules and from the procedures are direct calls to Oracle Data Integrator methods implemented in Java. These methods are usually used to generate some text that corresponds to the metadata stored into the Oracle Data Integrator repository.

2.2.1 Generic Syntax

The substitution methods are used in any text of a task of a Knowledge Module or of a procedure.

They can be used within any text using the following syntax

```
<%=java_expression%>
```

In this syntax:

- The `<%= %>` tags are used to output the text returned by `java_expression`. This syntax is very close to the syntax used in Java Server Pages (JSP).
- `Java expression` is any Java expression that returns a string.

The following syntax performs a call to the `getTable` method of the `odiRef` java object using three parameters. This method call returns a string. That is written after the `CREATE TABLE` text.

```
CREATE TABLE <%=odiRef.getTable("L", "INT_NAME", "A")%>
```

The Oracle Data Integrator Substitution API is implemented in the Java class `OdiReference`, whose instance `OdiRef` is available at any time. For example, to call a method called `getFrom()`, you have to write `odiRef.getFrom()`.

Note: For backward compatibility, the "**odiRef**" API can also be referred to as "**snpRef**" API. "snpRef" and "odiRef" object instances are synonyms, and the legacy syntax syntax `snpRef.<method_name>` is still supported but deprecated.

2.2.2 Specific Syntax for CKM

The following syntax is used in an IKM to call the execution of a check procedure (CKM).

This syntax automatically includes all the CKM procedure commands at this point of in the processing.

```
<% @ INCLUDE (CKM_FLOW | CKM_STATIC) [DELETE_ERROR] %>
```

The options for this syntax are:

- `CKM_FLOW`: triggers a flow control, according to the CKM choices made in the Control tab of the Interface.
- `CKM_STATIC`: Triggers a static control of the target datastore. Constraints defined for the datastore and selected as Static constraints will be checked.
- `DELETE_ERRORS`: This option causes automatic suppression of the errors detected.

For example: the following call triggers a flow control with error deletion.

```
<% @ INCLUDE  CKM_FLOW DELETE_ERROR %>
```

2.2.3 Using Flexfields

Flexfields are user-defined fields enabling to customize the properties of Oracle Data Integrator' objects. Flexfields are defined on the **Flexfield** tab of the object window and can be set for each object instance through the **Flexfield** tab of the object window.

When accessing an object properties through Oracle Data Integrator' substitution methods, if you specify the flexfield **Code**, Oracle Data Integrator will substitute the **Code** by the flexfield value for the object instance.

For instance:

`<%=odiRef.getTable("L", "MY_DATASTORE_FIELD", "W")%>` will return the value of the flexfield `MY_DATASTORE_FIELD` for the current table.

`<%=odiRef.getSrcTableList("", "[MY_DATASTORE_FIELD] ", ", ", "")%>` will return the flexfield value for each of the source tables of the interface.

It is also possible to get the value of a flexfield through the getFlexFieldValue() method.

Note: Flexfields exist only for certain object types. Objects that do not have a **Flexfield** tab do not support flexfields.

2.3 Using Substitution Methods in Actions

An action corresponds to a DDL operation (create table, drop reference, etc) used to generate a procedure to implement in a database the changes performed in a data integrator model (Generate DDL operation). Each action contains several **Action**

Lines, corresponding to the commands required to perform the DDL operation (for example, dropping a table requires dropping all its constraints first).

2.3.1 Action Lines Code

Action lines contain statements valid for the technology of the action group. Unlike procedures or knowledge module commands, these statements use a single connection (SELECT ... INSERT statements are not possible). In the style of the knowledge modules, action make use of the substitution methods to make their DDL code generic.

For example, an action line may contain the following code to drop a check constraint on a table:

```
ALTER TABLE <%=odiRef.getTable("L", "TARG_NAME", "A") %>
DROP CONSTRAINT <%=odiRef.getCK("COND_NAME") %>
```

2.3.2 Action Calls Method

The Action Calls methods are usable in the action lines only. Unlike other substitution methods, they are not used to generate text, but to generate actions appropriate for the context.

For example, to perform the a Drop Table DDL operation, we must first drop all foreign keys referring to the table.

In the *Drop Table* action, the first action line will use the dropReferringFKs() action call method to automatically generate a *Drop Foreign Key* action for each foreign key of the current table. This call is performed by creating an action line with the following code:

```
<% odiRef.dropReferringFKs(); %>
```

The syntax for calling the action call methods is:

```
<% odiRef.method_name(); %>
```

Note: The action call methods must be alone in an action line, should be called without a preceding "=" sign, and require a trailing semi-colon.

The following Action Call Methods are available for Actions:

- **addAKs()**: Call the *Add Alternate Key* action for all alternate keys of the current table.
- **dropAKs()**: Call the *Drop Alternate Key* action for all alternate keys of the current table.
- **addPK()**: Call the *Add Primary Key* for the primary key of the current table.
- **dropPK()**: Call the *Drop Primary Key* for the primary key of the current table.
- **createTable()**: Call the *Create Table* action for the current table.
- **dropTable()**: Call the *Drop Table* action for the current table.
- **addFKs()**: Call the *Add Foreign Key* action for all the foreign keys of the current table.

- **dropFKs()**: Call the *Drop Foreign Key* action for all the foreign keys of the current table.
- **enableFKs()**: Call the *Enable Foreign Key* action for all the foreign keys of the current table.
- **disableFKs()**: Call the *Disable Foreign Key* action for all the foreign keys of the current table.
- **addReferringFKs():** Call the *Add Foreign Key* action for all the foreign keys pointing to the current table.
- **dropReferringFKs():** Call the *Drop Foreign Key* action for all the foreign keys pointing to the current table.
- **enableReferringFKs():** Call the *Enable Foreign Key* action for all the foreign keys pointing to the current table.
- **disableReferringFKs()**: Call the *Disable Foreign Key* action for all the foreign keys pointing to the current table.
- **addChecks()**: Call the *Add Check Constraint* action for all check constraints of the current table.
- **dropChecks()**: Call the *Drop Check Constraint* action for all check constraints of the current table.
- **addIndexes()**: Call the *Add Index* action for all the indexes of the current table.
- **dropIndexes()**: Call the *Drop Index* action for all the indexes of the current table.
- **modifyTableComment():** Call the *Modify Table Comment* for the current table.
- **AddColumnsComment():** Call the *Modify Column Comment* for all the columns of the current table.

2.4 Working with Object Names

When working in Designer, you should avoid specifying physical information such as the database name or schema name as they may change depending on the execution context. The correct physical information will be provided by Oracle Data Integrator at execution time.

The substitution API has methods that calculate the fully qualified name of an object or datastore taking into account the context at runtime. These methods are listed in the table below:

Qualified Name Required	Method	Usable In
Any object named OBJ_NAME	getObjectName("L", "OBJ_NAME", "D")	Anywhere
The target datastore of the current interface	getTable("L", "TARG_NAME", "A")	LKM, CKM, IKM, JKM
The integration (I$) table of the current interface.	getTable("L", "INT_NAME", "A")	LKM, IKM
The loading table (C$) for the current loading phase.	getTable("L", "COLL_NAME", "A")	LKM
The error table (E$) for the datastore being checked.	getTable("L", "ERR_NAME", "A")	LKM, CKM, IKM
The datastore being checked	getTable("L", "CT_NAME", "A")	CKM

Qualified Name Required	Method	Usable In
The datastore referenced by a foreign key	getTable("L", "FK_PK_TABLE_NAME", "A")	CKM

2.5 Working with Lists of Tables, Columns and Expressions

Generating code from a list of items often requires a "while" or "for" loop. Oracle Data Integrator addresses this issue by providing powerful methods that help you generate code based on lists. These methods act as "iterators" to which you provide a substitution mask or pattern and a separator and they return a single string with all patterns resolved separated by the separator.

All of them return a string and accept at least these 4 parameters:

- **Start**: a string used to start the resulting string.
- **Pattern**: a substitution mask with attributes that will be bound to the values of each item of the list.
- **Separator**: a string used to separate each substituted pattern from the following one.
- **End**: a string appended to the end of the resulting string

Some of them accept an additional parameter (the **Selector**) that acts as a filter to retrieve only part of the items of the list. For example, list only the *mapped* column of the target datastore of an interface.

Some of these methods are summarized in the table below:

Method	Description	Usable In
getColList()	The most frequently-used method in Oracle Data Integrator. It returns a list of columns and expressions that need to be executed in the context where used. You can use it, for example, to generate lists like these: ■ Columns in a CREATE TABLE statement ■ Columns of the update key ■ Expressions for a SELECT statement in a LKM, CKM or IKM ■ Field definitions for a loading script This method accepts a "selector" as a 5th parameter to let you filter items as desired.	LKM, CKM, IKM, JKM, SKM
getTargetColList()	Returns the list of columns in the target datastore. This method accepts a selector as a 5th parameter to let you filter items as desired.	LKM, CKM, IKM, JKM,SKM
getAKColList()	Returns the list of columns defined for an alternate key.	CKM, SKM
getPKColList()	Returns the list of columns in a primary key. You can alternatively use getColList with the selector parameter set to "PK" .	CKM,SKM
getFKColList()	Returns the list of referencing columns and referenced columns of the current foreign key.	CKM,SKM

Method	Description	Usable In
getSrcTablesList()	Returns the list of source tables of an interface. Whenever possible, use the getFrom method instead. The getFrom method is discussed below.	LKM, IKM
getFilterList()	Returns the list of filter expressions in an interface. The getFilter method is usually more appropriate.	LKM, IKM
getJoinList()	Returns the list of join expressions in an interface. The getJoin method is usually more appropriate.	LKM, IKM
getGrpByList()	Returns the list of expressions that should appear in the group by clause when aggregate functions are detected in the mappings of an interface. The getGrpBy method is usually more appropriate.	LKM, IKM
getHavingList()	Returns the list of expressions that should appear in the having clause when aggregate functions are detected in the filters of an interface. The getHaving method is usually more appropriate.	LKM, IKM
getSubscriberList()	Returns a list of subscribers.	JKM

The following section provide examples illustrating how these methods work for generating code:

2.5.1 Using getTargetColList to create a table

The following example shows how to use a column list to create a table.

The following KM code:

```
Create table MYTABLE
<%=odiRef.getTargetColList("(\n", "\t[COL_NAME] [DEST_WRI_DT]", ",\n", "\n)")%>
```

Generates the following statement:

```
Create table MYTABLE
(
   CUST_ID numeric(10),
   CUST_NAME varchar(250),
   ADDRESS varchar(250),
   CITY varchar(50),
   ZIP_CODE varchar(12),
   COUNTRY_ID varchar(3)
)
```

In this example:

- **Start** is set to "(\n": The generated code will start with a parenthesis followed by a carriage return (\n).
- **Pattern** is set to "\t[COL_NAME] [DEST_WRI_DT]": The generated code will loop over every target column and generate a tab character (\t) followed by the column name ([COL_NAME]), a white space and the destination writable data type ([DEST_WRI_DT]).

- The **Separator** is set to ",\n": Each generated pattern will be separated from the next one with a comma (,) and a carriage return (\n)
- **End** is set to "\n)": The generated code will end with a carriage return (\n) followed by a parenthesis.

2.5.2 Using getColList in an Insert values statement

The following example shows how to use column listing to insert values into a table.

For following KM code:

```
insert into MYTABLE
(
<%=odiRef.getColList("", "[COL_NAME]", ", ", "\n", "INS AND NOT TARG")%>
<%=odiRef.getColList(",", "[COL_NAME]", ", ", "", "INS AND TARG")%>
)
Values
(
<%=odiRef.getColList("", ":[COL_NAME]", ", ", "\n", "INS AND NOT TARG")%>
<%=odiRef.getColList(", ", "[EXPRESSION]", ", ", "", "INS AND TARG")%>
)
```

Generates the following statement:

```
insert into MYTABLE
(
CUST_ID, CUST_NAME, ADDRESS, CITY, COUNTRY_ID
, ZIP_CODE, LAST_UPDATE
)
Values
(
:CUST_ID, :CUST_NAME, :ADDRESS, :CITY, :COUNTRY_ID
, 'ZZ2345', current_timestamp
)
```

In this example, the values that need to be inserted into MYTABLE are either bind variables with the same name as the target columns or constant expressions if they are executed on the target. To obtain these 2 distinct set of items, the list is split using the Selector parameter:

- "INS AND NOT TARG": first, generate a comma-separated list of columns ([COL_NAME]) mapped to bind variables in the "value" part of the statement (:[COL_NAME]). Filter them to get only the ones that are flagged to be part of the INSERT statement and that are **not executed on the target.**
- "INS AND TARG": then generate a comma separated list of columns ([COL_NAME]) corresponding to expression ([EXPRESSION]) that are flagged to be part of the INSERT statement and that are **executed on the target**. The list should start with a comma if any items are found.

2.5.3 Using getSrcTableList

The following example concatenates the list of the source tables of an interface for logging purposes.

For following KM code:

```
insert into MYLOGTABLE
(
INTERFACE_NAME,
```

```
DATE_LOADED,
SOURCE_TABLES
)
values
(
'<%=odiRef.getPop("POP_NAME")%>',
current_date,
'' <%=odiRef.getSrcTablesList("|| ", "'[RES_NAME]'", " || ',' || ", "")%>
)
```

Generates the following statement:

```
insert into MYLOGTABLE
(
INTERFACE_NAME,
DATE_LOADED,
SOURCE_TABLES
)
values
(
'Int. CUSTOMER',
current_date,
'' || 'SRC_CUST' || ',' || 'AGE_RANGE_FILE' || ',' || 'C$0_CUSTOMER'
)
```

In this example, getSrcTableList generates a message containing the list of resource names used as sources in the interface to append to MYLOGTABLE. The separator used is composed of a concatenation operator (||) followed by a comma enclosed by quotes (',') followed by the same operator again. When the table list is empty, the SOURCE_TABLES column of MYLOGTABLE will be mapped to an empty string ('').

2.6 Generating the Source Select Statement

LKMs and IKMs both manipulate a source result set. For the LKM, this result set represents the pre-transformed records according to the mappings, filters and joins that need to be executed on the source. For the IKM, however, the result set represents the transformed records matching the mappings, filters and joins executed on the staging area.

To build these result sets, you will usually use a SELECT statement in your KMs. Oracle Data Integrator has some advanced substitution methods, including getColList, that help you generate this code:

Method	Description	Usable In
getFrom()	Returns the FROM clause of a SELECT statement with the appropriate source tables, left, right and full outer joins. This method uses information from the topology to determine the SQL capabilities of the source or target technology. The FROM clause is built accordingly with the appropriate keywords (INNER, LEFT etc.) and parentheses when supported by the technology. ■ When used in an LKM, it returns the FROM clause as it should be executed by the source server. ■ When used in an IKM, it returns the FROM clause as it should be executed by the staging area server.	LKM, IKM

Method	Description	Usable In
getFilter()	Returns filter expressions separated by an "AND" operator. ■ When used in an LKM, it returns the filter clause as it should be executed by the source server. ■ When used in an IKM, it returns the filter clause as it should be executed by the staging area server.	LKM, IKM
getJrnFilter()	Returns the special journal filter expressions for the journalized source datastore. This method should be used with the CDC framework.	LKM, IKM
getGrpBy()	Returns the GROUP BY clause when aggregation functions are detected in the mappings. The GROUP BY clause includes all mapping expressions referencing columns that do not contain aggregation functions. The list of aggregation functions are defined by the language of the technology in the topology.	LKM, IKM
getHaving()	Returns the HAVING clause when aggregation functions are detected in filters. The having clause includes all filters expressions containing aggregation functions. The list of aggregation functions are defined by the language of the technology in the topology.	LKM, IKM

To obtain the result set from any SQL RDBMS source server, you would use the following SELECT statement in your LKM:

```
select <%=odiRef.getPop("DISTINCT_ROWS")%>
<%=odiRef.getColList("", "[EXPRESSION]\t[ALIAS_SEP] [CX_COL_NAME]", ",\n\t", "",
"")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getJoin()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

To obtain the result set from any SQL RDBMS staging area server to build your final flow data, you would use the following SELECT statement in your IKM. Note that the getColList is filtered to retrieve only expressions that are not executed on the target and that are mapped to writable columns.

```
select <%=odiRef.getPop("DISTINCT_ROWS")%>
<%=odiRef.getColList("", "[EXPRESSION]", ",\n\t", "", "(not TRG) and REW")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getJrnFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

As all filters and joins start with an AND, the WHERE clause of the SELECT statement starts with a condition that is always true (1=1).

2.7 Working with Datasets

Oracle Data Integrator supports datasets. Each dataset represents a group of joined and filtered sources tables, with their mappings. Datasets are merged into the target datastore using set-based operators (UNION, INTERSECT, etc) at the integration phase.

During the loading phase, the LKM always works on one dataset. Duringthe integration phase, when all datasets need to merged, certain odiRef APIs that support working on a specific dataset are called using an index that identifies the dataset.

The following example explains how this dataset merging is done.

```
<%for (int i=0; i < odiRef.getDataSetCount(); i++){%>
<%=odiRef.getDataSet(i, "Operator")%>
select <%=odiRef.getUserExit("OPTIMIZER_HINT")%>
<%=odiRef.getPop("DISTINCT_ROWS")%>
<%=odiRef.getColList(i,"", "[EXPRESSION]", ",\n\t", "", "(((INS or UPD) and !TRG)
and REW)")%>,
<% if (odiRef.getDataSet(i, "HAS_JRN").equals("1")) { %>
JRN_FLAG IND_UPDATE
<%} else {%>
'I' IND_UPDATE
<%}%>
from <%=odiRef.getFrom(i)%>
where(1=1)
<%=odiRef.getJoin(i)%>
<%=odiRef.getFilter(i)%>
<%=odiRef.getJrnFilter(i)%>
<%=odiRef.getGrpBy(i)%>
<%=odiRef.getHaving(i)%>
<%}%>
```

A Java For loop iterates over the datasets. The number of datasets is retrieved using the getDataSetCount method. For each dataset, a SELECT statement is issued, each statement separated from the previous one by the dataset's set-based operator retrieved using the getDataSet method.

The select statement is build as in Generating the Source Select Statement, except that each method call is parameterized with i, the index of the dataset being processed. For example, getFrom(i) generates the from statement for the dataset identified by the value of i.

All the methods that support a parameter for the dataset index also support a syntax without this index value. Outside an IKM, then should be used without the dataset index. Within an IKM, if used without the dataset index, these method address the first dataset. This syntax is backward compatible with previous Oracle Data Integrator interfaces and knowledge modules.

2.8 Obtaining Other Information with the API

The following methods provide additional information which may be useful:

Method	Description	Usable In
getPop()	Returns information about the current interface.	IKM, LKM, CKM, JKM
getInfo()	Returns information about the source or target server.	Any procedure or KM
getSession()	Returns information about the current running session	Any procedure or KM
getOption()	Returns the value of a particular option	Any procedure or KM
getFlexFieldValue()	Returns information about a flex field value. Not that with the "List" methods, flex field values can be specified as part of the pattern parameter.	Any procedure or KM
getJrnInfo()	Returns information about the CDC framework	JKM, LKM, IKM
getTargetTable()	Returns information about the target table of an interface	LKM, IKM, CKM
getModel()	Returns information about the current model during a reverse-engineering process.	RKM
getPop()	Returns information about the current interface.	LKM, IKM

2.9 Advanced Techniques for Code Generation

You can use conditional branching and advanced programming techniques to generate code. The code generation in Oracle Data Integrator is able to interpret any Java code enclosed between "<%" and "%>" tags.

The following examples illustrate how you can use these advanced techniques.

Using Java Variables and String Functions

The following KM Code creates a string variable and uses it in a substitution method call :

```
<%
String myTableName;
myTableName = "ABCDEF";
%>
drop table <%=odiRef.getObjectName(myTableName.toLowerCase())%>
```

Generates the following:

```
drop table SCOTT.abcdef
```

Using a KM Option to Generate Code Conditionally

The following KM code generates code depending on the OPT001 option value.

```
<%
String myOptionValue=odiRef.getOption("OPT001");
if (myOption.equals("TRUE"))
{
out.print("/* Option OPT001 is set to TRUE */");
}
else
{
%>
/* The OPT001 option is not properly set */
```

```
<%
}
%>
```

If OPT001 is set to TRUE, then the following is generated:

```
/* Option OPT001 is set to TRUE */
```

Otherwise the following is generated

```
/* The OPT001 option is not set to TRUE */
```

3

Reverse-Engineering Strategies

This chapter explains the customized reverse-engineering process and the strategies used in the Reverse-engineering Knowledge Modules for retrieving advanced metadata.

This chapter contains the following sections:

- Section 3.1, "Customized Reverse-Engineering Process"
- Section 3.2, "Case Studies"

3.1 Customized Reverse-Engineering Process

Oracle Data Integrator Standard Reverse-Engineering relies on the capabilities of the driver used to connect a given data server to return rich metadata describing the data strucuture.

When this metadata is not accurate, or needs to be enriched with some metadata retrieved from the data server, customized reverse-engineering can be used.

3.1.1 SNP_REV tables

The Oracle Data Integrator repository contains a set of metadata staging tables, called the SNP_REV tables.

These SNP_REV tables content is managed using the following tools:

- OdiReverseResetTable resets the content of these tables for a given model.
- OdiReverseGetMetadata populates these tables using a process similar to the standard JDBC reverse-engineering.
- OdiReverseSetMetadata applies the content of these staging tables to the repository tables describing the datastores, columns, constraints, etc. This action modifies the Oracle Data Integrator model.

See Appendix B, "SNP_REV Tables Reference" for a reference of the SNP_REV table, and the *Developer's Guide for Oracle Data Integrator* for more information for a reference of the reverse-engineering tools.

3.1.2 Customized Reverse-Engineering Strategy

Customized Reverse-Engineering strategy follows a pattern common to all RKMs.

This patterns includes the following steps:

1. Call the OdiReverseResetTable tool to reset the SNP_REV tables from previous executions.

2. Load the SNP_REV tables. This is performed using three main patterns:
   - Retrieve metadata from the metadata provider and load them into to SNP_ REV tables. This is the pattern used for example in the *RKM Oracle*.
   - Retreive metadata from a third party provider. This is the pattern used for example in the *RKM File (FROM EXCEL)*. Metadata is not extracted from the files described in the model but from a Micrisift Excel Excel spreadsheet that contains the description of these files.
   - Pre-populate the SNP_REV tables using OdiReverseGetMetadata and then fix/enrich this metadata using queries targeting these tables.
3. Call the OdiReverseSetMetaData tool to apply the changes to the current Oracle Data Integrator model.

In an RKM, the source and target commands work are follow:

- The **Command on Target** specified with an *Undefined* technology on the *Autocommit* transaction targets the SNP_REV tables in the repository.
- The **Command on Source** specified with an *Undefined* Schema on the *Autocommit* transaction retreives data from the data-server containing the data structure to reverse-engineer. If you want to use a metadata provider (for example an Excel spreadsheet), you must specify a specific technology and logical schema.
- Calls to Tools (such as OdiReverseSetMetadata) are specified in the **Command on Target**, with the *ODI Tools* technology.

3.2 Case Studies

This section provides examples of reverse-egineering strategies.

3.2.1 RKM Oracle

The RKM Oracle is a typical example of a reverse-engineering process using a database dictionnary as the metadata provider.

The commands below are extracted from the RKM for Oracle and provided as examples. You can review the code of this knowledge module by editing it in Oracle Data Intagrator Studio.

3.2.1.1 Reset SNP_REV Tables

This task resets the content of the SNP_REV tables for the current model.

Command on Target (ODI Tools)

```
OdiReverseResetTable -MODEL=<%=odiRef.getModel("ID")%>
```

3.2.1.2 Get Tables

This task retreives the list of tables from the Oracle system tables and loads this content into the SNP_REV tables.

Command on Source

```
Select
t.TABLE_NAMETABLE_NAME,
t.TABLE_NAMERES_NAME,
replace(t.TABLE_NAME,'<%=odiRef.getModel("REV_ALIAS_LTRIM")%>','')TABLE_ALIAS,
substr(tc.COMMENTS,1,250)TABLE_DESC,
```

```
'T' TABLE_TYPE,
t.NUM_ROWSR_COUNT,
SUBSTR(PARTITIONING_TYPE ,1,1) PARTITIONING_TYPE,
SUBSTR(SUBPARTITIONING_TYPE,1,1) SUBPARTITIONING_TYPE
FromALL_TABLESt,
ALL_TAB_COMMENTS tc,
ALL_PART_TABLES tp
Where ...
...
```

Command on Target

```
insert into SNP_REV_TABLE
(
I_MOD,
TABLE_NAME,
RES_NAME,
TABLE_ALIAS,
TABLE_TYPE,
TABLE_DESC,
IND_SHOW,
R_COUNT,
PARTITION_METH,
SUB_PARTITION_METH
)
values
(
<%=odiRef.getModel("ID")%>,
:TABLE_NAME,
:RES_NAME,
:TABLE_ALIAS,
'T',
:TABLE_DESC,
'1',
:R_COUNT,
:PARTITIONING_TYPE,
:SUBPARTITIONING_TYPE

)
```

3.2.1.3 Get views, partitions, columns, FK, Keys and other Oracle Metadata

Subsequent commands use the same pattern to load the SNP_REV tables from the content of the Oracle system tables.

3.2.1.4 Set Metadata

This task resets the content of the SNP_REV tables for the current model.

Command on Target (ODI Tools)

```
OdiReverseSetMetaData -MODEL=<%=odiRef.getModel("ID")%>
```

4

Data Integrity Strategies

This chapter explains the data integrity strategies used for performing flow and static checks. These strategies are implemented in the Check Knowledge Modules.

This chapter contains the following sections:

- Section 4.1, "Data Integrity Check Process"
- Section 4.2, "Case Studies"

4.1 Data Integrity Check Process

Data Integrity Check Process checks is activated in the following cases:

- When a **Static Control** is started (from Studio, or using a package) on a model, sub-model or datastore. The data in the datastores are checked against the constraints defined in the Oracle Data Integrator model.
- If an interface is executed and a **Flow Control** is activated in the IKM. The flow data staged in the integration table (I$) is checked against the constraints of the target datastore, as defined in the model. Only those of the constraints selected in the interface are checked.

In both those cases, a CKM is in charge of checking the data quality of data according to a predefined set of constraints. The CKM can be used either to check existing data when used in a "static control" or to check flow data when used in a "flow control". It is also in charge of removing the erroneous records from the checked table if specified.

In the case of a static control, the CKM used is defined in the model. In the case of a flow control, it is specified for the interface.

4.1.1 Check Knowledge Module Overview

Standard CKMs maintain 2 different types of tables:

- A single summary table named *SNP_CHECK_TAB* for each data server, created in the work schema of the default physical schema of the data server. This table contains a summary of the errors for every table and constraint. It can be used, for example, to analyze the overall data quality of a model.
- An error table named *E$_<datastore name>* for every datastore that was checked. The error table contains the actual records rejected by data quality processes (static and flow controls) launched for this table.

A standard CKM is composed of the following steps:

- Drop and create the summary table. The DROP statement is executed only if the designer requires it for resetting the summary table. The CREATE statement is always executed but the error is tolerated if the table already exists.
- Remove the summary records from the previous run from the summary table
- Drop and create the error table. The DROP statement is executed only if the designer requires it for recreating the error table. The CREATE statement is always executed but error is tolerated if the table already exists.
- Remove rejected records from the previous run from the error table
- Reject records that violate the primary key constraint.
- Reject records that violate any alternate key constraint
- Reject records that violate any foreign key constraint
- Reject records that violate any check condition constraint
- Reject records that violate any mandatory column constraint
- Remove rejected records from the checked table if required
- Insert the summary of detected errors in the summary table.

CKM commands should be tagged to indicate how the code should be generated. The tags can be:

- "Primary Key": The command defines the code needed to check the primary key constraint
- "Alternate Key": The command defines the code needed to check an alternate key constraint. During code generation, Oracle Data Integrator will use this command for every alternate key
- "Join": The command defines the code needed to check a foreign key constraint. During code generation, Oracle Data Integrator will use this command for every foreign key
- "Condition": The command defines the code needed to check a condition constraint. During code generation, Oracle Data Integrator will use this command for every check condition
- "Mandatory": The command defines the code needed to check a mandatory column constraint. During code generation, Oracle Data Integrator will use this command for mandatory column
- "Remove Errors": The command defines the code needed to remove the rejected records from the checked table.

4.1.2 Error Tables Structures

This section describes the typical structure of the Error and Summary Tables.

4.1.2.1 Error Table Structure

The E$ error table has the list of columns described in the following table:

Columns	Description
[Columns of the checked table]	The error table contains all the columns of the checked datastore.

Columns	Description
ERR_TYPE	Type of error: ■ 'F' when the datastore is checked during flow control ■ 'S' when the datastore is checked using static control
ERR_MESS	Error message related to the violated constraint
CHECK_DATE	Date and time when the datastore was checked
ORIGIN	Origin of the check operation. This column is set either to the datastore name or to an interface name and ID depending on how the check was performed.
CONS_NAME	Name of the violated constraint.
CONS_TYPE	Type of the constraint: ■ 'PK': Primary Key ■ 'AK': Alternate Key ■ 'FK': Foreign Key ■ 'CK': Check condition ■ 'NN': Mandatory column

4.1.2.2 Summary Table Structure

The SNP_CHECK table has the list of columns described in the following table:

Column	Description
ODI_CATALOG_ NAME	Catalog name of the checked table, where applicable
ODI_SCHEMA_NAME	Schema name of the checked table, where applicable
ODI_RESOURCE_ NAME	Resource name of the checked table
ODI_FULL_RES_ NAME	Fully qualified name of the checked table. For example <catalog>.<schema>.<table>
ODI_ERR_TYPE	Type of error: ■ 'F' when the datastore is checked during flow control ■ 'S' when the datastore is checked using static control
ODI_ERR_MESS	Error message
ODI_CHECK_DATE	Date and time when the datastore was checked
ODI_ORIGIN	Origin of the check operation. This column is set either to the datastore name or to an interface name and ID depending on how the check was performed.
ODI_CONS_NAME	Name of the violated constraint.
ODI_CONS_TYPE	Type of constraint: ■ 'PK': Primary Key ■ 'AK': Alternate Key ■ 'FK': Foreign Key ■ 'CK': Check condition ■ 'NN': Mandatory column (Not Null)
ODI_ERR_COUNT	Total number of records rejected by this constraint during the check process
ODI_SESS_NO	ODI session number

Column	Description
ODI_PK	Unique identifier for this table, where appicable

4.2 Case Studies

This section provides examples of data integrity check strategies.

4.2.1 Oracle CKM

The CKM Oracle is a typical example of a data integrity check.

The commands below are extracted from the CKM for Oracle and provided as examples. You can review the code of this knowledge module by editing it in Oracle Data Integrator Studio.

4.2.1.1 Drop Check Table

This task drops the error summary table. This command runs only if the DROP_CHECK_TABLE is set to Yes, and has the *Ignore Errors* flag activated. It will not stop the CKM if the summary table is not found.

Command on Target (Oracle)

```
drop table <%=odiRef.getTable("L","CHECK_NAME","W")%> <% if (new
Integer(odiRef.getOption( "COMPATIBLE" )).intValue() >= 10 ) { out.print( "purge"
); }; %>
```

4.2.1.2 Create Check Table

This task creates the error summary table. This command always runs and has the *Ignore Errors* flag activated. It will not stop the CKM if the summary table already exist.

Command on Target (Oracle)

```
...
create table <%=odiRef.getTable("L","CHECK_NAME","W")%>
(
CATALOG_NAME <%=odiRef.getDataType("DEST_VARCHAR", "100", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%> ,
SCHEMA_NAME <%=odiRef.getDataType("DEST_VARCHAR", "100", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%> ,
RESOURCE_NAME <%=odiRef.getDataType("DEST_VARCHAR", "100", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
FULL_RES_NAME <%=odiRef.getDataType("DEST_VARCHAR", "100", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ERR_TYPE <%=odiRef.getDataType("DEST_VARCHAR", "1", "")%> <%=odiRef.getInfo("DEST_
DDL_NULL")%>,
ERR_MESS <%=odiRef.getDataType("DEST_VARCHAR", "250", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%> ,
CHECK_DATE <%=odiRef.getDataType("DEST_DATE", "", "")%> <%=odiRef.getInfo("DEST_
DDL_NULL")%>,
ORIGIN <%=odiRef.getDataType("DEST_VARCHAR", "100", "")%> <%=odiRef.getInfo("DEST_
DDL_NULL")%>,
CONS_NAME <%=odiRef.getDataType("DEST_VARCHAR", "35", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
CONS_TYPE <%=odiRef.getDataType("DEST_VARCHAR", "2", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ERR_COUNT <%=odiRef.getDataType("DEST_NUMERIC", "10", "")%>
```

```
<%=odiRef.getInfo("DEST_DDL_NULL")%>
)
...
```

4.2.1.3 Create Error Table

This task creates the error (E$) table. This command always runs and has the *Ignore Errors* flag activated. It will not stop the CKM if the error table already exist.

Note the use of the *getCollist* method to add the list of columns from the checked to this table structure.

Command on Target (Oracle)

```
...
create table <%=odiRef.getTable("L","ERR_NAME", "W")%>
(
ODI_ROW_ID UROWID,
ODI_ERR_TYPE <%=odiRef.getDataType("DEST_VARCHAR", "1", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_ERR_MESS <%=odiRef.getDataType("DEST_VARCHAR", "250", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_CHECK_DATE <%=odiRef.getDataType("DEST_DATE", "", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
<%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_
NULL"), ",\n\t", "", "")%>,
ODI_ORIGIN <%=odiRef.getDataType("DEST_VARCHAR", "100", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_CONS_NAME <%=odiRef.getDataType("DEST_VARCHAR", "35", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_CONS_TYPE <%=odiRef.getDataType("DEST_VARCHAR", "2", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_PK <%=odiRef.getDataType("DEST_VARCHAR", "32", "")%> PRIMARY KEY,
ODI_SESS_NO <%=odiRef.getDataType("DEST_VARCHAR", "19", "")%>
)
...
```

4.2.1.4 Insert PK Errors

This task inserts into the error (E$) table the errors detected while checking a primary key. This command always runs, has the **Primary Key** checkbox active and has **Log Counter** set to *Error* to count these records as errors.

> **Note:** When using a CKM to perform flow control from an interface, you can define the maximum number of errors allowed. This number is compared to the total number of records returned by every command in the CKM of which the **Log Counter** is set to *Error*.

Note the use of the *getCollist* method to insert into the error table the whole record being checked and the use of the *getPK* and *getInfo* method to retrieve contextual information.

Command on Target (Oracle)

```
insert into <%=odiRef.getTable("L","ERR_NAME", "W")%>
(
ODI_PK,
```

```
ODI_SESS_NO,
ODI_ROW_ID,
ODI_ERR_TYPE,
ODI_ERR_MESS,
ODI_ORIGIN,
ODI_CHECK_DATE,
ODI_CONS_NAME,
ODI_CONS_TYPE,
<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "MAP")%>
)
selectSYS_GUID(),
<%=odiRef.getSession("SESS_NO")%>,
rowid,
'<%=odiRef.getInfo("CT_ERR_TYPE")%>',
'<%=odiRef.getPK("MESS")%>',
'<%=odiRef.getInfo("CT_ORIGIN")%>',
<%=odiRef.getInfo("DEST_DATE_FCT")%>,
'<%=odiRef.getPK("KEY_NAME")%>',
'PK',
<%=odiRef.getColList("", odiRef.getTargetTable("TABLE_ALIAS")+".[COL_NAME]",
",\n\t", "", "MAP")%>
from <%=odiRef.getTable("L", "CT_NAME", "A")%> <%=odiRef.getTargetTable("TABLE_
ALIAS")%>
where exists  (
select <%=odiRef.getColList("", "SUB.[COL_NAME]", ",\n\t\t\t", "", "PK")%>
from <%=odiRef.getTable("L","CT_NAME","A")%> SUB
where <%=odiRef.getColList("", "SUB.[COL_NAME]="+odiRef.getTargetTable("TABLE_
ALIAS")+".[COL_NAME]", "\n\t\t\tand ", "", "PK")%>
group by <%=odiRef.getColList("", "SUB.[COL_NAME]", ",\n\t\t\t", "", "PK")%>
having count(1) > 1
)
<%=odiRef.getFilter()%>
```

4.2.1.5 Delete Errors from Controlled Table

This task removed from the controlled table (static control) or integration table (flow control) the rows detected as erroneous.

This task is always executed and has the **Remove Errors** option selected.

Command on Target (Oracle)

```
delete from<%=odiRef.getTable("L", "CT_NAME", "A")%>  T
where exists (
select 1
from <%=odiRef.getTable("L","ERR_NAME", "W")%> E
where ODI_SESS_NO = <%=odiRef.getSession("SESS_NO")%>
and T.rowid = E.ODI_ROW_ID
)
```

4.2.2 Dynamically Create Non-Existing References

The following use case describes an example of customization that can be performed on top of an existing CKM.

4.2.2.1 Use Case

When loading a data warehouse, you may have records referencing data from other tables, but the referenced records do not yet exist.

Suppose, for example, that you receive daily sales transactions records that reference product SKUs. When a product does not exist in the products table, the default behavior of the standard CKM is to reject the sales transaction record into the error table instead of loading it into the data warehouse. However, to meet the requirements of your project you want to load this sales record into the data warehouse and create an empty product on the fly to ensure data consistency. The data analysts would then simply analyze the error tables and complete the missing information for products that were automatically added to the products table.

The following sequence illustrates this example:

1. The source flow data is staged by the IKM in the "I$_SALES" table to load the SALES table. The IKM calls the CKM to have it check the data quality.
2. The CKM checks every constraint including the FK_SALES_PRODUCTS foreign key defined between the target SALES table and the PRODUCTS Table. It rejects record with SALES_ID='4' in the error table as referenced product with PRODUCT_ID="P25" doesn't exist in the products table.
3. The CKM automatically inserts the missing "P25" product reference in the products table and assigns an '<*unknown*>' value to the PRODUCT_NAME. All other columns are set to null or default values.
4. The CKM does not remove the rejected record from the source flow I$ table, as it became consistent
5. The IKM writes the flow data to the target

In the sequence above, steps 3 and 4 differ from the standard CKM and need to be customized.

4.2.2.2 Discussion

To implement such a CKM, you will notice that some information is missing in the Oracle Data Integrator default metadata. We would need the following:

- A new flexfield called REF_TAB_DEF_COL on the Reference object containing the column of the referenced table that must be populated with the '<*unknown*>' value (PRODUCT_NAME, in our case)
- A new column (ODI_AUTO_CREATE_REFS) in the error table to indicate whether an FK error needs to automatically create the missing reference or not. This flag will be populated while detecting the FK errors.
- A new flexfields called AUTO_CREATE_REFS on the "Reference" object, that will state whether a constraint should automatically cause missing references creation. See the *Developer's Guide for Oracle Data Integrator* for more information about Flex Fields.

Now that we have all the required metadata, we can start enhancing the default CKM to meet our requirements. The steps of the CKM will therefore be (changes are highlighted in bold font):

- Drop and create the summary table.
- Remove the summary records of the previous run from the summary table
- Drop and create the error table. **Add the extra ODI_AUTO_CREATE_REFS column to the error table.**
- Remove rejected records from the previous run from the error table
- Reject records that violate the primary key constraint.

- Reject records that violate each alternate key constraint
- Reject records that violate each foreign key constraint, **and store the value of the AUTO_CREATE_REFS flexfield in the ODI_AUTO_CREATE_REFS column.**
- **For every foreign key error detected, if the ODI_AUTO_CREATE_REFS is set to "yes", insert missing references in the referenced table**.
- Reject records that violate each check condition constraint
- Reject records that violate each mandatory column constraint
- Remove rejected records from the checked table if required. **Do not remove records for which the constraint behavior is set to Yes**
- Insert the summary of detected errors in the summary table.

4.2.2.3 Implementation Details

The following command modifications are performed to implement the required changes to the CKM. The changes are highlighted in bold in the code.

4.2.2.3.1 Create Errors Table

The task is modified to create the new *ODI_AUTO_CREATE_REFS* column into the error table.

Command on Target (Oracle)

```
...
create table <%=odiRef.getTable("L","ERR_NAME", "W")%>
(
ODI_AUTO_CREATE_REFS <%=odiRef.getDataType("DEST_VARCHAR", "3", "")%>
ODI_ROW_ID UROWID,
ODI_ERR_TYPE <%=odiRef.getDataType("DEST_VARCHAR", "1", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_ERR_MESS <%=odiRef.getDataType("DEST_VARCHAR", "250", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_CHECK_DATE <%=odiRef.getDataType("DEST_DATE", "", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
<%=odiRef.getColList("", "[COL_NAME]\t[DEST_WRI_DT] " + odiRef.getInfo("DEST_DDL_
NULL"), ",\n\t", "", "")%>,
ODI_ORIGIN <%=odiRef.getDataType("DEST_VARCHAR", "100", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_CONS_NAME <%=odiRef.getDataType("DEST_VARCHAR", "35", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_CONS_TYPE <%=odiRef.getDataType("DEST_VARCHAR", "2", "")%>
<%=odiRef.getInfo("DEST_DDL_NULL")%>,
ODI_PK <%=odiRef.getDataType("DEST_VARCHAR", "32", "")%> PRIMARY KEY,
ODI_SESS_NO <%=odiRef.getDataType("DEST_VARCHAR", "19", "")%>
)
...
```

4.2.2.3.2 Insert FK Errors

The task is modified to take into account the new *ODI_AUTO_CREATE_REFS* column and load it with the content of the flexfield defined on the FK to indicate whether this constraint should automatically create missing references. Note the use of the *getFK* method to retrieve the value of the *AUTO_CREATE_REFS* flexfield.

Command on Target (Oracle)

```
...
insert into <%=odiRef.getTable("L","ERR_NAME", "W")%>
(
ODI_AUTO_CREATE_REFS,
ODI_PK,
ODI_SESS_NO,
ODI_ROW_ID,
ODI_ERR_TYPE,
ODI_ERR_MESS,
ODI_CHECK_DATE,
ODI_ORIGIN,
ODI_CONS_NAME,
ODI_CONS_TYPE,
<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "MAP")%>
)
select
'<%=odiRef.getFK("AUTO_CREATE_REFS")%>',
SYS_GUID(),
<%=odiRef.getSession("SESS_NO")%>,
rowid,
...
```

4.2.2.3.3 Insert Missing References

The new task is added after the *insert FK errors* task. It has the **Join** option checked.

Note the following:

- The *getFK("AUTO_CREATE_FS")* method is used to retrieve the AUTO_CREATE_FS flexfield value that conditions the generation of the SQL statement.
- The *getFK("REF_TAB_DEF_COL")* method is used to retrieve from the flexfield the name of the column to set to *'<undefined>'*.
- The *getFKColList* method is used to retrieve the list of columns participating to the foreign key and create the missing reference primary key columns content.
- The filter made to retrieve only the records corresponding to the current checked foreign key constraint with the AUTO_CREATE_REFS flag set to Yes.

Command on Target (Oracle)

```
<% if (odiRef.getFK("AUTO_CREATE_REFS").equals("Yes")) { %>

insert into <%=odiRef.getTable("L", "FK_PK_TABLE_NAME", "A")%>
(
<%=odiRef.getFKColList("", "[PK_COL_NAME]", ",", "")%>,
<%=odiRef.getFK("REF_TAB_DEF_COL")%>
)

select distinct
<%=odiRef.getFKColList("", "[COL_NAME]", ",", "")%>,
'<UNKNOWN>'
from <%=odiRef.getTable("L","ERR_NAME", "A")%>
where
CONS_NAME = '<%=odiRef.getFK("FK_NAME")%>'
And CONS_TYPE = 'FK'
And ORIGIN = '<%=odiRef.getInfo("CT_ORIGIN")%>'
And AUTO_CREATE_REFS = 'Yes'
```

```
<%}%>
```

4.2.2.3.4 Delete Errors from Controlled Table

This task is modified to avoid deleting the foreign key records for which a reference have been created. These can remain in the controlled table.

Command on Target (Oracle)

```
delete from<%=odiRef.getTable("L", "CT_NAME", "A")%>  T
where exists (
select 1
from <%=odiRef.getTable("L","ERR_NAME", "W")%> E
where ODI_SESS_NO = <%=odiRef.getSession("SESS_NO")%>
and T.rowid = E.ODI_ROW_ID
and E.AUTO_CREATE_REFS <> 'Yes'
)
```

5 Loading Strategies

This chapter explains the loading strategies used for loading data into the staging area. These strategies are implemented in the Loading Knowledge Modules.

This chapter contains the following sections:

- Section 5.1, "Loading Process"
- Section 5.2, "Case Studies"

5.1 Loading Process

A loading process is required when source data needs to be loaded into the staging area. This loading is needed when some transformation take place in the staging area and the source schema is not located in the same server as the staging area. The staging area is the target of the loading phase.

5.1.1 Loading Process Overview

A typical loading process works in the following way:

1. A temporary *loading table* is dropped (if it exists) and then created in the staging area
2. Data is loaded from the source into this loading table using a *loading method*.

   Action 1 and 2 are repeated for all the source data that needs to be moved to the staging area.

   The data is used in the integration phase to load the integration table.
3. After the integration phase, before the interface completes, the temporary loading table is dropped.

5.1.2 Loading Table Structure

The loading process creates in the staging area a loading table. This loading table is typically prefixed with a `C$`.

A loading table represent a *source set* and not a *source datastore.* There is no direct mapping between the sources datastore and the loading table. Source sets appear in the flow tab of the interface editor.

The following cases illustrate the notion of source set:

- If a source CUSTOMER table has only 2 column CUST_NAME, CUST_ID used in mapping and joins on the staging area, then the loading table will only contain an

image of these two columns. Columns not needed for the rest of the integration flow not appear the loading table.

- If a is CUSTOMER table is filtered on CUST_AGE on the source, and CUST_AGE is not used afterwards, then the loading table will not include CUST_AGE. The loading process will process the filter in the source data server, and the loading table will contain the filtered records.
- If two table CUSTOMER and SALES_REPS are combined using a join on the source and the resulting source set is used in transformations taking place in the staging area, the loading table will contain the combined columns from CUSTOMER and SALES_REPS.
- If **all** the columns of a source datastore are mapped and this datastore is not joined on the source, then the source set is the whole source datastore. In that case, the loading table is the exact image of the source datastore. This is the case for source technologies with no transformation capabilities such as File.

5.1.3 Loading Method

The loading method is the key to optimal performance when loading data from a source to the staging area. There are several loading methods, which can be grouped in the following categories:

- Loading Using the Agent
- Loading File Using Loaders
- Loading Using Unload/Load
- Loading Using RDBMS-Specific Strategies

5.1.3.1 Loading Using the Agent

The run-time agent is able to read a result set using JDBC on a source server and write this result set using JDBC to the loading table in the staging area. To use this method, the knowledge module needs to include a command with a SELECT on the source with a corresponding INSERT on the target.

This method may not be suited for large volumes as data is read row-by-row in arrays, using the array fetch feature, and written row-by-row, using the batch update feature.

5.1.3.2 Loading File Using Loaders

When the interface contains a flat file as a source, you may want to use a strategy that leverages the most efficient loading utility available for the staging area technology, rather than the standard LKM File to SQL that uses the ODI built-in driver for files. Most RDBMSs have fast loading utilities to load flat files into tables, such as Oracle's SQL*Loader, Microsoft SQL Server bcp, Teradata FastLoad or MultiLoad.

Such LKM will load the source file into the staging area, and all transformations will take place in the staging area afterwards.

A typical LKM using a loading utility will include the following sequence of steps:

1. Drop and create the loading table in the staging area
2. Generate the script required by the loading utility to load the file to the loading table.
3. Execute the appropriate operating system command to start the load and check its return code.

4. Possibly analyze any log files produced by the utility for error handling.
5. Drop the loading table once the integration phase has completed.

5.1.3.3 Loading Using Unload/Load

When the source result set is on a remote database server, an alternate solution to using the agent to transfer the data is to unload it to a file and then load that file into the staging area.

This is usually the most efficient method when dealing with large volumes across heterogeneous technologies. For example, you can unload data from a Microsoft SQL Server source using bcp and load this data into an Oracle staging area using SQL*Loader.

The steps of LKMs that follow this strategy are often as follows:

1. Drop and create the loading table in the staging area
2. Unload the data from the source to a temporary flat file using either a source database unload utility (such as Microsoft SQL Server bcp or DB2 unload) or the built-in OdiSqlUnload tool.
3. Generate the script required by the loading utility to load the temporary file to the loading table.
4. Execute the appropriate operating system command to start the load and check its return code.
5. Possibly analyze any log files produced by the utility for error handling.
6. Drop the loading table once the integration KM has terminated, and drop the temporary file.

When using an unload/load strategy, data needs to be staged twice: once in the temporary file and a second time in the loading table, resulting in extra disk space usage and potential efficiency issues. A more efficient alternative would be to use pipelines between the "unload" and the "load" utility. Unfortunately, not all the operating systems support file-based pipelines (FIFOs).

5.1.3.4 Loading Using RDBMS-Specific Strategies

Certain RDBMSs have a mechanism for transferring data across servers. For example:

- Oracle: database links
- Microsoft SQL Server: linked servers
- IBM DB2 400: DRDA file transfer

Other databases implement specific mechanisms for loading files into a table, such as Oracle's External Table feature.

These loading strategies are implemented into specific KM that create the appropriate objects (views, dblinks, etc.) and implement the appropriate commands for using these features.

5.2 Case Studies

This section provides example of loading strategies.

5.2.1 LKM SQL to SQL

The LKM SQL to SQL is a typical example of the loading phase using the agent.

The commands below are extracted from this KM and are provided as examples. You can review the code of this knowledge module by editing it in Oracle Data Integrator Studio.

5.2.1.1 Drop Work Table

This task drops the loading table. This command is always executed and has the *Ignore Errors* flag activated. It will not stop the LKM if the loading table is not found.

Command on Target

```
drop table <%=snpRef.getTable("L", "COLL_NAME", "A")%>
```

5.2.1.2 Create Work Table

This task drops the loading table. This command is always executed.

Note the use of the property *COLL_NAME* of the *getTable* method that returns the name of the loading table.

Command on Target

```
create table <%=snpRef.getTable("L", "COLL_NAME", "A")%>
(
<%=snpRef.getColList("", "[CX_COL_NAME]\t[DEST_WRI_DT] " + snpRef.getInfo("DEST_
DDL_NULL"), ",\n\t", "","")%>
)
```

5.2.1.3 Load Data

This task reads data on the source connection and loads it into the loading table. This command is always executed.

> **Note:** The loading phase is always using auto commit, as ODI temporary tables do not contain unrecoverable data.

Command on Source

Note the use of the *getFilter*, *getJoin*, *getFrom*, etc. methods. these methods are shortcuts that return contextual expressions. For example, *getFilter* returns all the filter expressions executed on the source. Note also the use of the *EXPRESSION* property of *getColList*, that will return the source columns and the expressions executed on the source. These expressions and source columns are aliases after *CX_COL_NAME*, which is the name of their corresponding column in the loading table.

This select statement will cause the correct transformation (mappings, joins, filters, etc.) to be executed by the source engine.

```
select<%=snpRef.getPop("DISTINCT_ROWS")%>
<%=snpRef.getColList("", "[EXPRESSION]\t[ALIAS_SEP] [CX_COL_NAME]", ",\n\t", "",
"")%>
from<%=snpRef.getFrom()%>
where(1=1)
<%=snpRef.getFilter()%>
<%=snpRef.getJrnFilter()%>
<%=snpRef.getJoin()%>
<%=snpRef.getGrpBy()%>
```

```
<%=snpRef.getHaving()%>
```

Command on Target

Note here the use of the biding using `:[CX_COL_NAME]`. The *CX_COL_NAME* binded value will match the alias on the source column.

```
insert into <%=snpRef.getTable("L", "COLL_NAME", "A")%>
(
<%=snpRef.getColList("", "[CX_COL_NAME]", ",\n\t", "","")%>
)
values
(
<%=snpRef.getColList("", ":[CX_COL_NAME]", ",\n\t", "","")%>
)
```

5.2.1.4 Drop Work Table

This task drops the loading table. This command is executed if the DELETE_ TEMPORARY_OBJECTS knowledge module option is selected. This option will allow to preserve the loading table for debugging.

Command on Target

```
drop table <%=snpRef.getTable("L", "COLL_NAME", "A")%>
```

6 Integration Strategies

This chapter explains the integration strategies used in integration interfaces. These strategies are implemented in the Integration Knowledge Modules.

This chapter contains the following sections:

- Section 6.1, "Integration Process"
- Section 6.2, "Case Studies"

6.1 Integration Process

An integration process is always needed in an interface. This process integrates data from the source or loading tables into the target datastore, using a temporary integration table.

An integration process uses an integration strategy which defines the steps required in the integration process. Example of integration strategies are:

- *Append*: Optionally delete all records in the target datastore and insert all the flow into the target.
- *Control Append*: Optionally delete all records in the target datastore and insert all the flow into the target. This strategy includes an optional flow control.
- *Incremental Update*: Optionally delete all records in the target datastore. Identify new and existing records by comparing the flow with the target, then insert new records and update existing records in the target. This strategy includes an optional flow control.
- *Slowly Changing Dimension*: Implement a Type 2 Slowly Changing Dimension, identifying fields that require a simple update in the target record when change, fields that require to historize the previous record state.

This phase may involve one single server, when the staging area and the target are located in the same data server, on two servers when the staging area and target are on different servers.

6.1.1 Integration Process Overview

The integration process depends strongly on the strategy being used.

The following elements are used in the integration process:

- An *integration table* (also known as the *flow table*) is sometimes needed to stage data after all staging area transformation are made. This loading table is named after the target table, prefixed with I$. This integration table is the image of the target table with extra fields required for the strategy to be implemented. The data in this

table is flagged, transformed or checked before being integrated into the target table.

- The source and/or loading tables (created by the LKM). The integration process loads data from these tables into the integration table or directly into the target tables.
- Check Knowledge Module. The IKM may initiate a flow check phase to check the data in the integration table against some of the constraints of the target table. Invalid data is removed from the integration table (*removed from the flow*).
- *Interface metadata*, such as Insert, Update, UD1, etc., or *model metadata* such as the Slowly Changing Dimension behavior are used at integration phase to parameterize column-level behavior in the integration strategies.

A typical integration process works in the following way:

1. Create a temporary integration table if needed. For example, an update flag taking value I or U to identify which of the rows are to be inserted or updated.
2. Load data from the source and loading tables into this integration table, executing those of the transformations (joins, filters, mapping) specified on the staging area.
3. Perform some transformation on the integration table to implement the integration strategy. For example, compare the content of the integration table with the target table to set the update flag.
4. Modify the content Load data from the integration table into the target table.

6.1.2 Integration Strategies

The following sections explain some of the integration strategies used in Oracle Data Integrator. They are grouped into two families:

- Strategies with Staging Area on the Target
- Strategies with the Staging Area Different from the Target.

6.1.2.1 Strategies with Staging Area on the Target

These strategies are used when the staging area schema is located in the same data server as the target table schema. In this configuration, complex integration strategies can take place

6.1.2.1.1 Append

This strategy simply inserts the incoming data flow into the target datastore, possibly deleting the content of the target beforehand.

This integration strategy includes the following steps:

1. Delete (or truncate) all records from the target table. This step usually depends on a KM option.
2. Transform and insert data from sources located on the same server and from loading tables in the staging area. When dealing with remote source data, LKMs will have already prepared loading tables. Sources on the same server can be read directly. The integration operation will be a direct INSERT/SELECT statement leveraging containing all the transformations performed on the staging area in the SELECT clause and on all the transformation on the target in the INSERT clause.

3. Commit the Transaction. The operations performed on the target should be done within a transaction and committed after they are all complete. Note that committing is typically triggered by a KM option called *COMMIT*.

The same integration strategy can be obtained by using the *Control Append* strategy and not choosing to activate flow control.

6.1.2.1.2 Control Append

In the Append strategy, flow data is simply inserted in the target table without any flow control. This approach can be improved by adding extra steps that will store the flow data in an integration table ("I$"), then call the CKM to isolate erroneous records in the error table ("E$").

This integration strategy includes the following steps:

1. Drop (if it exists) and create the integration table in the staging area. This is created with the same columns as the target table so that it can be passed to the CKM for flow control.
2. Insert data in the loading table from the sources and loading tables using a single INSERT/SELECT statement similar to the one loading the target in the append strategy.
3. Call the CKM for flow control. The CKM will evaluate every constraint defined for the target table on the integration table data. It will create an error table and insert the erroneous records into this table. It will also remove erroneous records from the integration table.

   After the CKM completes, the integration table will only contain valid records. Inserting them in the target table can then be done safely.
4. Remove all records from the target table. This step can be made dependent on an option value set by the designer of the interface
5. Append the records from the integration table to the target table in a single INSERT/SELECT statement.
6. Commit the transaction.
7. Drop the temporary integration table.

Error Recycling

In some cases, it is useful to recycle errors from previous runs so that they are added to the flow and applied again to the target. This method can be useful for example when receiving daily sales transactions that reference product IDs that may not exist. Suppose that a sales record is rejected in the error table because the referenced product ID does not exist in the product table. This happens during the first run of the interface. In the meantime the missing product ID is created by the data administrator. Therefore the rejected record becomes valid and should be re-applied to the target during the next execution of the interface.

This mechanism is implement IKMs by an extra task that inserts all the rejected records of the previous executions of this interface from the error table into integration table. This operation is made prior to calling the CKM to check the data quality, and is conditioned by a KM option usually called *RECYCLE_ERRORS*.

6.1.2.1.3 Incremental Update

The Incremental Update strategy is used to integrate data in the target table by comparing the records of the flow with existing records in the target according to a set

of columns called the "update key". Records that have the same update key are updated when their associated data is not the same. Those that don't yet exist in the target are inserted. This strategy is often used for dimension tables when there is no need to keep track of the records that have changed.

The challenge with such IKMs is to use set-oriented SQL based programming to perform all operations rather than using a row-by-row approach that often leads to performance issues. The most common method to build such strategies often relies on the integration table ("I$") which stores the transformed source sets. This method is described below:

1. Drop (if it exists) and create the integration table in the staging area. This is created with the same columns as the target table so that it can be passed to the CKM for flow control. It also contains an `IND_UPDATE` column that is used to flag the records that should be inserted ("I") and those that should be updated ("U").
2. Transform and insert data in the loading table from the sources and loading tables using a single INSERT/SELECT statement. The `IND_UPDATE` column is set by default to "I".
3. Recycle the rejected records from the previous run to the integration table if the *RECYCLE_ERROR KM* option is selected.
4. Call the CKM for flow control. The CKM will evaluate every constraint defined for the target table on the integration table data. It will create an error table and insert the erroneous records into this table. It will also remove erroneous records from the integration table.
5. Update the integration table to set the `IND_UPDATE` flag to "U" for all the records that have the same update key values as the target ones. Therefore, records that already exist in the target will have a "U" flag. This step is usually an UPDATE/SELECT statement.
6. Update the integration table again to set the `IND_UPDATE` column to "N" for all records that are already flagged as "U" and for which the column values are exactly the same as the target ones. As these flow records match exactly the target records, they don't need to be used to update the target data.

   After this step, the integration table is ready for applying the changes to the target as it contains records that are flagged:

   - "I": these records should be inserted into the target.
   - "U": these records should be used to update the target.
   - "N": these records already exist in the target and should be ignored.
7. Update the target with records from the integration table that are flagged "U". Note that the update statement is typically executed prior to the INSERT statement to minimize the volume of data manipulated.
8. Insert records in the integration table that are flagged "I" into the target.
9. Commit the transaction.
10. Drop the temporary integration table.

Optimization

This approach can be optimized depending on the underlying database. The following examples illustrate such optimizations:

- With Teradata, it may be more efficient to use a left outer join between the flow data and the target table to populate the integration table with the `IND_UPDATE` column already set properly.
- With Oracle, it may be more efficient in some cases to use a MERGE INTO statement on the target table instead of an UPDATE then INSERT.

Update Key

The update key should always be unique. In most cases, the primary key will be used as an update key. The primary key cannot be used, however, when it is automatically calculated using an increment such as an identity column, a rank function, or a sequence. In this case an update key based on columns present in the source must be used.

Comparing Nulls

When comparing data values to determine what should not be updated, the join between the integration table and the target table is expressed on each column as follow:

<target_table>.ColumnN = <loading_table>.ColumnN or (<target_table> is null and <loading_table>.ColumnN is null)

This is done to allow comparison between null values, so that a null value matches another null value. A more elegant way of writing it would be to use the coalesce function. Therefore the WHERE predicate could be written this way:

<%=odiRef.getColList("","coalesce(" + odiRef.getTable("L", "INT_NAME", "A") + ".[COL_NAME], 0) = coalesce(T.[COL_NAME], 0)", " \nand\t", "", "((UPD and !TRG) and !UK) ")%>

Column-Level Insert/Update Behavior

Columns updated by the UPDATE statement are not the same as the ones used in the INSERT statement. The UPDATE statement uses selector "UPD and not UK" to filter only mappings marked as "Update" in the interface and that do not belong to the update key. The INSERT statement uses selector "INS" to retrieve mappings marked as "insert" in the interface.

Transaction

It is important that the UPDATE and the INSERT statements on the target belong to the same transaction. Should any of them fail, no data will be inserted or updated in the target.

6.1.2.1.4 Slowly Changing Dimensions

Type 2 Slowly Changing Dimension (SCD) is a strategy used for loading data warehouses. It is often used for loading dimension tables, in order to keep track of changes on specific columns. A typical slowly changing dimension table would contain the flowing columns:

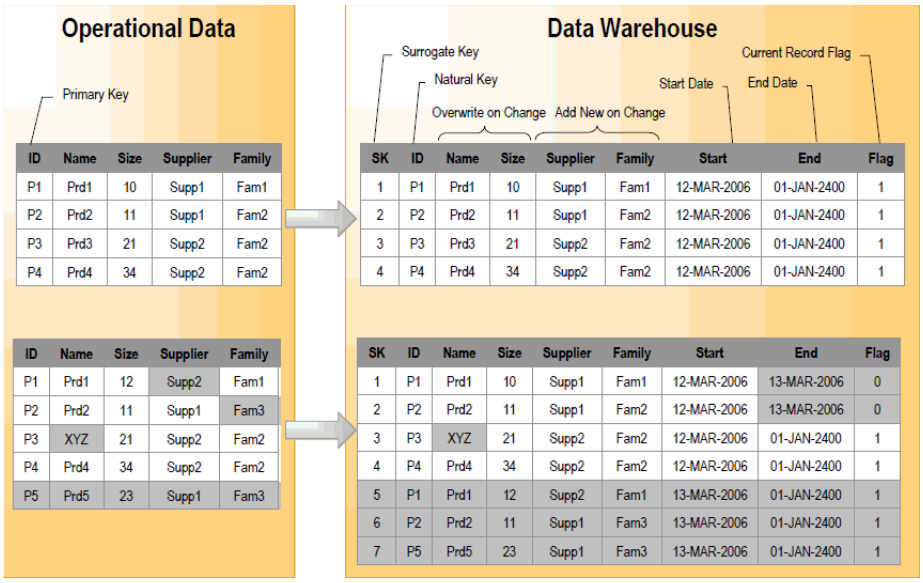
- A *surrogate key*. This is usually a numeric column containing an automatically-generated number (using an identity column, a rank function or a sequence).
- A *natural key*. This is the list of columns that represent the primary key of the operational system.
- Columns that one must *overwritten on change*.

- Columns that require to *add row on change*.
- A *starting timestamp* column indicating when the record was created in the data warehouse
- An *ending timestamp* column indicating when the record became obsolete (closing date)
- A *current record flag* indicating whether the record is the actual one (1) or an old one (0)

The following example illustrate the Slowly Changing Dimension behavior.

In the operational system, a product is defined by its ID that acts as a primary key. Every product has a name, a size, a supplier and a family. In the Data Warehouse a new version of this product is stored whenever the supplier or the family is updated in the operational system.

Figure 6–1 Type 2 Slow Changing Dimensions Example



ID	Name	Size	Supplier	Family
P1	Prd1	10	Supp1	Fam1
P2	Prd2	11	Supp1	Fam2
P3	Prd3	21	Supp2	Fam2
P4	Prd4	34	Supp2	Fam2

SK	ID	Name	Size	Supplier	Family	Start	End	Flag
1	P1	Prd1	10	Supp1	Fam1	12-MAR-2006	01-JAN-2400	1
2	P2	Prd2	11	Supp1	Fam2	12-MAR-2006	01-JAN-2400	1
3	P3	Prd3	21	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1
4	P4	Prd4	34	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1

ID	Name	Size	Supplier	Family
P1	Prd1	12	Supp2	Fam1
P2	Prd2	11	Supp1	Fam3
P3	XYZ	21	Supp2	Fam2
P4	Prd4	34	Supp2	Fam2
P5	Prd5	23	Supp1	Fam3

SK	ID	Name	Size	Supplier	Family	Start	End	Flag
1	P1	Prd1	10	Supp1	Fam1	12-MAR-2006	13-MAR-2006	0
2	P2	Prd2	11	Supp1	Fam2	12-MAR-2006	13-MAR-2006	0
3	P3	XYZ	21	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1
4	P4	Prd4	34	Supp2	Fam2	12-MAR-2006	01-JAN-2400	1
5	P1	Prd1	12	Supp2	Fam1	13-MAR-2006	01-JAN-2400	1
6	P2	Prd2	11	Supp1	Fam3	13-MAR-2006	01-JAN-2400	1
7	P5	Prd5	23	Supp1	Fam3	13-MAR-2006	01-JAN-2400	1

In this example, the product dimension is first initialized in the Data Warehouse on March 12, 2006. All the records are inserted and are assigned a calculated surrogate key as well as a fake ending date set to January 1, 2400. As these records represent the current state of the operational system, their current record flag is set to 1. After the first load, the following changes happen in the operational system:

1. The supplier is updated for product P1
2. The family is updated for product P2
3. The name is updated for product P3
4. Product P5 is added

These updates have the following impact on the data warehouse dimension:

- The update of the supplier of P1 is translated into the creation of a new current record (Surrogate Key 5) and the closing of the previous record (Surrogate Key 1)
- The update of the family of P2 is translated into the creation of a new current record (Surrogate Key 6) and the closing of the previous record (Surrogate Key 2)
- The update of the name of P3 simply updates the target record with Surrogate Key 3
- The new product P5 is translated into the creation of a new current record (Surrogate Key 7).

To create a Knowledge Module that implements this behavior, it is necessary to know which columns act as a surrogate key, a natural key, a start date etc. Oracle Data Integrator stores this information in **Slowly Changing Dimension Behavior** field in the **Description** tab for every column in the model.

When populating such a datastore in an interface, the IKM has access to this metadata using the *SCD_xx* selectors on the *getColList()* substitution method.

The way Oracle Data Integrator implements Type 2 Slowly Changing Dimensions is described below:

1. Drop (if it exists) and create the integration table in the staging area.
2. Insert the flow data in the integration table using only mappings that apply to the *natural key*, *overwrite on change* and *add row on change* columns. Set the *starting timestamp* to the current date and the *ending timestamp* to a constant.
3. Recycle previous rejected records
4. Call the CKM to perform a data quality check on the flow
5. Flag the records in the integration table to 'U' when the *natural key* and the *add row on change* columns have not changed compared to the current records of the target.
6. Update the target with the columns flagged *overwrite on change* by using the integration table content filtered on the 'U' flag.
7. Close old records - those for which the natural key exists in the integration table, and set their *current record flag* to 0 and their *ending timestamp* to the current date
8. Insert the new changing records with their *current record flag* set to 1
9. Drop the integration table.

Again, this approach can be adapted. There may be some cases where the SQL produced requires further tuning and optimization.

6.1.2.2 Strategies with the Staging Area Different from the Target

These strategies are used when the staging area cannot be located on the same data server as the target datastore. This configuration is mainly used for data servers with no transformation capabilities (Files, for example). In this configuration, only simple integration strategies are possible

6.1.2.2.1 File to Server Append

There are some cases when the source is a single file that can be loaded directly into the target table using the most efficient method. By default, Oracle Data Integrator suggests to locate the staging area on the target server, use a LKM to stage the source file in a loading table and then use an IKM to integrate the loaded data to the target table.

If the source data is not transformed, the loading phase is not necessary.

In this situation you would use an IKM that directly loads the file data to the target: This requires setting the staging area on the source file logical schema. By doing this, Oracle Data Integrator will automatically suggest to use a "Multi-Connection" IKM that moves data between a remote staging area and the target.

Such an IKM would use a loader, and include the following steps:

1. Generate the appropriate load utility script
2. Run the loader utility

An example of such KM is the IKM File to Teradata (TTU).

6.1.2.2.2 Server to Server Append

When using a staging area different from the target and when setting this staging area to an RDBMS, it is possible to use an IKM that moves the transformed data from the staging area to the remote target. This type of IKM is similar to a LKM and follows the same rules.

The steps when using the agent are usually:

1. Delete (or truncate) all records from the target table. This step usually depends on a KM option.
2. Insert the data from the staging area to the target. This step has a SELECT statement in the "Command on Source" tab that will be executed on the staging area. The INSERT statement is written using bind variables in the "Command on Target" tab and will be executed for every batch on the target table.

   The IKM SQL to SQL Append is a typical example of such KM.

   Variation of this strategy use loaders or database specific methods for loading data from the staging area to the target instead of the agent.

6.1.2.2.3 Server to File or JMS Append

When the target datastore is a file or JMS queue or topic the staging area is set on a different location than the target. Therefore, if you want to target a file or queue datastore you will have to use a "Multi-Connection" IKM that will integrate the transformed data from your staging area to this target. The method to perform this data movement depends on the target technology. For example, it is possible to use the agent or specific features of the target (such as a Java API)

Typical steps of such an IKM will include:

- Reset the target file or queue made dependent on an option
- Unload the data from the staging area to the file or queue

6.2 Case Studies

This section provides example of integration strategies and customizations.

6.2.1 Simple Replace or Append

The simplest strategy for integrating data in an existing target table, provided that all source data is already in the staging area is to replace and insert the records in the target. Therefore, the simplest IKM would be composed of 2 steps:

- Remove all records from the target table. This step can be made dependent on an option set by the designer of the interface

- Transform and insert source records from all datasets. When dealing with remote source data, LKMs will have already prepared loading tables with pre-transformed result sets. If the interface uses source data sets on the same server as the target (and the staging area as well), they will be joined to the other loading tables. Therefore the integration operation will be a straight INSERT/SELECT statement leveraging all the transformation power of the target Teradata box.

The following example gives you the details of these steps:

6.2.1.1 Delete Target Table

This task deletes the data from the target table. This command runs in a transaction and is not committed. It is executed if the DELETE_ALL Knowledge Module option is selected.

Command on Target

```
delete from <%=odiRef.getTable("L","INT_NAME","A")%>
```

6.2.1.2 Insert New Rows

This task insert rows from the staging table into the target table. This command runs in the same transaction as all operations made on the target and is not committed. A final Commit transaction command triggers the commit on the target.

Note that this commands selects the data from the different datasets defined for the interface. Using a for loop, it goes through all the datasets, generates for each dataset a SELECT query. These queries are merged using set-based operations (UNION, INTERSECT, etc.) and the resulting data flow is inserted into the target table.

Command on Target

```
insert into<%=odiRef.getTable("L","TARG_NAME","A")%>
(
<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and REW)")%>
<%=odiRef.getColList(",", "[COL_NAME]", ",\n\t", "", "((INS and TRG) and REW)")%>
)

select
    <%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "", "((INS and !TRG) and
REW)")%>
  <%=odiRef.getColList(",", "[EXPRESSION]", ",\n\t", "", "((INS and TRG) and
REW)")%>
FROM (
<%for (int i=0; i < odiRef.getDataSetCount(); i++){%>
<%=odiRef.getDataSet(i, "Operator")%>
select <%=odiRef.getPop("DISTINCT_ROWS")%>
<%=odiRef.getColList(i,"", "[EXPRESSION] [COL_NAME]", ",\n\t", "", "((INS and
!TRG) and REW)")%>
from<%=odiRef.getFrom(i)%>
where<% if (odiRef.getDataSet(i, "HAS_JRN").equals("1")) { %>
JRN_FLAG <> 'D'<%} else {%>(1=1)<% } %>
<%=odiRef.getJoin(i)%>
<%=odiRef.getFilter(i)%>
<%=odiRef.getJrnFilter(i)%>
<%=odiRef.getGrpBy(i)%>
<%=odiRef.getHaving(i)%>
<%}%>
)
```

6.2.2 Backup the Target Table Before Loading

A project requirements is to backup every data warehouse table prior to loading the current data. This can help restoring the data warehouse to its previous state in case of a major problem. The backup tables are called like the data table with a "_BCK" suffix.

A first solution to this requirement would be to develop interfaces that would duplicate data from every target datastore to its corresponding backup one. These interfaces would be triggered prior to the ones that would populate the data warehouse. Unfortunately, this solution would lead to significant development and maintenance effort as it requires the creation of an additional interface for every target datastore. The number of interfaces to develop and maintain would be at least doubled!

A simple solution would be to implement this behavior in the IKM used to populate the target datastores. This would be done using a single CREATE AS SELECT statement that creates and populates to the backup table right before modifying the target. Therefore, the backup operation becomes automatic and the developers would no longer need to worry about it.

This example shows how this behavior could be implemented in the IKM Oracle Incremental Update.

Before the Update Existing Rows and Insert New Rows tasks that modify the target, the following tasks are added.

6.2.2.1 Drop Backup Table

This task drops the backup table.

Command on Target

```
Drop table <%=odiRef.getTable("L","TARG_NAME","A")%>_BCK
```

6.2.2.2 Create Backup Table

This task creates and populates the backup table.

Command on Target

```
Create table <%=odiRef.getTable("L","TARG_NAME","A")%>_BCK as
select <%=odiRef.getTargetColList("", "[COL_NAME]", ",", "")%>
from <%=odiRef.getTable("L","TARG_NAME","A")%>
```

6.2.3 Tracking Records for Regulatory Compliance

Some data warehousing projects could require keeping track of every insert or update operation done to target tables for regulatory compliance. This could help business analysts understand what happened to their data during a certain period of time.

Even if you can achieve this behavior by using the slowly changing dimension Knowledge Modules, it can also be done by simply creating a copy of the flow data before applying it to the target table.

Suppose that every target table has a corresponding tracking table with a "_RGG" suffix with all the data columns plus some additional regulatory compliance columns such as:

- The Job Id
- The Job Name
- Date and time of the operation

- The type of operation ("Insert" or "Update")

You would populate this table directly from the integration table after applying the inserts and updates to the target, and before the end of the IKM.

For example, in the case of the Oracle Incremental Update IKM, you would add the following tasks just after the Update Existing Rows and Insert New Rows tasks that modify the target.

6.2.3.1 Load Tracking Records

This task loads data in the tracking table.

Command on Target

```
insert into <%=odiRef.getTable("L","TARG_NAME","A")%>_RGC
(
JOBID,
JOBNAME,
OPERATIONDATE,
OPERATIONTYPE,
<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "")%>
)
select <%=odiRef.getSession("SESS_NO")%> /* JOBID */,
<%=odiRef.getSession("SESS_NAME")%> /* JOBNAME */,
Current_timestamp /* OPERATIONDATE */,
Case when IND_UPDATE = 'I' then 'Insert' else 'Update' end /* OPERATIONTYPE */,
<%=odiRef.getColList("", "[COL_NAME]", ",\n\t", "")%>
from <%=odiRef.getTable("L","INT_NAME","A")%>
where IND_UPDATE <> 'N'
```

This customization could be extended of course by creating automatically the tracking table using the IKM if it does not exist yet.

A

Substitution API Reference

This appendix provides a list of the Oracle Data Integrator odiRef API.

See Chapter 2, "Introduction to the Substitution API" for introductory information about using this API.

A.1 Substitution Methods List

The substitution are listed below depending on the type of knowledge module into which they can be used. The "Global Methods" list lists the methods that can be used in any situation.

Refer to the description of a given method itself for more information about its behavior in a given knowledge module or action.

This section contains the following topics:

- Global Methods
- Journalizing Knowledge Modules
- Loading Knowledge Modules
- Check Knowledge Modules
- Integration Knowledge Modules
- Reverse-Engineering Knowledge Modules
- Service Knowledge Modules
- Actions

A.1.1 Global Methods

The following methods can be used in all knowledge module and actions:

- getCatalogName() Method
- getCatalogNameDefaultPSchema() Method
- getColDefaultValue() Method
- getContext() Method
- getDataType() Method
- getFlexFieldValue() Method
- getInfo() Method
- getJDBCConnection() Method

- getJDBCConnectionFromLSchema() Method
- getNbInsert(), getNbUpdate(), getNbDelete(), getNbErrors() and getNbRows() Methods
- getObjectName() Method
- getObjectNameDefaultPSchema() Method
- getOption() Method
- getPackage() Method
- getPrevStepLog() Method
- getQuotedString() Method
- getSchemaName() Method
- getSchemaNameDefaultPSchema() Method
- getSession() Method
- getSessionVarList() Method
- getStep() Method
- getSysDate() Method
- setNbInsert(), setNbUpdate(), setNbDelete(), setNbErrors() and setNbRows() Methods
- setTaskName() Method

A.1.2 Journalizing Knowledge Modules

In addition to the methods from in the "Global Methods" list, the following methods can be used specifically in Journalizing Knowledge Modules (JKM):

- getJrnFilter() Method
- getJrnInfo() Method
- getSubscriberList() Method
- getTable() Method
- getColList() Method

A.1.3 Loading Knowledge Modules

In addition to the methods from in the "Global Methods" list, the following methods can be used specifically in Loading Knowledge Modules (LKM):

- getColList() Method
- getDataSet() Method
- getDataSetCount() Method
- getFilter() Method
- getFilterList() Method
- getFrom() Method
- getGrpBy() Method
- getGrpByList() Method

- getHaving() Method
- getHavingList() Method
- getJoin() Method
- getJoinList() Method
- getJrnFilter() Method
- getJrnInfo() Method
- getPop() Method
- getSrcColList() Method
- getSrcTablesList() Method
- getTable() Method
- getTargetColList() Method
- getTableName() Method
- getTargetTable() Method
- getTemporaryIndex() Method
- getTemporaryIndexColList() Method
- setTableName() Method

A.1.4 Check Knowledge Modules

In addition to the methods from in the "Global Methods" list, the following methods can be used specifically in Check Knowledge Modules (CKM):

- getAK() Method
- getAKColList() Method
- getCK() Method
- getColList() Method
- getFK() Method
- getFKColList() Method
- getNotNullCol() Method
- getPK() Method
- getPKColList() Method
- getPop() Method
- getTable() Method
- getTargetColList() Method
- getTargetTable() Method

A.1.5 Integration Knowledge Modules

In addition to the methods from in the "Global Methods" list, the following methods can be used specifically in Integration Knowledge Modules IKM):

- getColList() Method

- getDataSet() Method
- getDataSetCount() Method
- getFilter() Method
- getFilterList() Method
- getFrom() Method
- getGrpBy() Method
- getGrpByList() Method
- getHaving() Method
- getHavingList() Method
- getJoin() Method
- getJoinList() Method
- getJrnFilter() Method
- getJrnInfo() Method
- getPop() Method
- getSrcColList() Method
- getSrcTablesList() Method
- getTable() Method
- getTableName() Method
- getTargetColList() Method
- getTargetTable() Method
- getTemporaryIndex() Method
- getTemporaryIndexColList() Method
- setTableName() Method

A.1.6 Reverse-Engineering Knowledge Modules

In addition to the methods from in the "Global Methods" list, the following methods can be used specifically in Reverse-engineering Knowledge Modules (RKM):

- getModel() Method

A.1.7 Service Knowledge Modules

In addition to the methods from in the "Global Methods" list, the following methods can be used specifically in Service Knowledge Modules (SKM):

- hasPK() Method
- nextAK() Method
- nextCond() Method
- nextFK() Method

A.1.8 Actions

In addition to the methods from in the "Global Methods" list, the following methods can be used specifically in Actions.

- getAK() Method
- getAKColList() Method
- getCK() Method
- getColList() Method
- getColumn() Method
- getFK() Method
- getFKColList() Method
- getIndex() Method
- getIndexColList() Method
- getNewColComment() Method
- getNewTableComment() Method
- getPK() Method
- getPKColList() Method
- getTable() Method
- getTargetTable() Method
- isColAttrChanged() Method

A.2 Substitution Methods Reference

This section provides an alphabetical list of the substitution methods. Each method is detailed with usage, description, parameters and example code.

A.2.1 getAK() Method

Use to return information about an alternate key.

Usage

```
public java.lang.String getAK(java.lang.String pPropertyName)
```

Description

This method returns information relative to the alternate key of a datastore during a check procedure. It is only accessible from a Check Knowledge Module if the current task is tagged "alternate key".

In an action, this method returns information related to the alternate key currently handled by the DDL command.

Parameters

Parameter	Type	Description
pPropertyName	String	String containing the name of the requested property.

The following table lists the different possible values for pPropertyName.

Parameter Values	Description
ID	Internal number of the AK constraint.
KEY_NAME	Name of the alternate key
MESS	Error message relative to the constraint of the alternate key
FULL_NAME	Full name of the AK generated with the local object mask.
<flexfield code>	Value of the flexfield for this AK.

Examples

```
The alternate key of my table is named: <%=odiRef.getAK("KEY_NAME")%>
```

A.2.2 getAKColList() Method

Use to return information about the columns of an alternate key.

Usage

```
public java.lang.String getAKColList( java.lang.String pStart,
java.lang.String pPattern,java.lang.String pEnd)
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getAKColList(
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of columns and expressions for the alternate key currently checked.

The pPattern parameter is interpreted and then repeated for each element of the list. It is separated from its predecessor by the pSeparator parameter. The generated string starts with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains an element for each column of the current alternate key. It is accessible from a Check Knowledge Module if the current task is tagged as an "alternate key".

In an action, this method returns the list of the columns of the alternate key handled by the DDL command, ordered by their position in the key.

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameters	Type	Description
pStart	String	This sequence marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of attributes that can be used in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern sequence is replaced with its value. The attributes must be between brackets. ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This sequence marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as their associated description.

Parameter Value	Description
Parameter value	Description
I_COL	Column internal identifier
COL_NAME	Name of the key column
COL_HEADING	Header of the key column
COL_DESC	Column description
POS	Position of the column
LONGC	Length (Precision) of the column
SCALE	Scale of the column
FILE_POS	Beginning position of the column (fixed file)
BYTES	Number of physical bytes of the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column
COL_MANDATORY	Mandatory character of the column: ■ 0: null authorized ■ 1: non null
CHECK_FLOW	Flow control flag of the column: ■ 0: do not check ■ 1: check
CHECK_STAT	Static control flag of the column: ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column

Parameter Value	Description
COL_DEC_SEP	Decimal symbol for the column
REC_CODE_LIST	List of the record codes retained for the column
COL_NULL_IF_ERR	Processing flag for the column: ■ 0: Reject ■ 1: Set active trace to null ■ 2: Set inactive trace to null
DEF_VALUE	Default value for the column
EXPRESSION	Not used
CX_COL_NAME	Not used
ALIAS_SEP	Grouping symbol used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
<flexfield code>	Flexfield value for the current column.

Examples

If the CUSTOMER table has an alternate key AK_CUSTOMER (CUST_ID, CUST_ NAME) and you want to generate the following code:

```
create table T_AK_CUSTOMER
(CUST_ID numeric(10) not null, CUST_NAME varchar(50) not null)
```

You can use the following code:

```
create table T_<%=odiRef.getAK("KEY_NAME")%>
<%=odiRef.getAKColList("(", "[COL_NAME] [DEST_CRE_DT] not null", ", ", ")")%>
```

Explanation: the getAKColList function will be used to generate the `(CUST_ID numeric(10) not null, CUST_NAME varchar(50) not null)` part, which starts and stops with a parenthesis and repeats the pattern (column, a data type, and not null) separated by commas for each column of the alternate key. Thus

- the first parameter **"("** of the function indicates that we want to start the string with the string "("
- the second parameter **"[COL_NAME] [DEST_CRE_DT] not null"** indicates that we want to repeat this pattern for each column of the alternate key. The keywords [COL_NAME] and [DEST_CRE_DT] reference valid keywords of the Pattern Attributes List table
- the third parameter **", "** indicates that we want to separate interpreted occurrences of the pattern with the string ", "

- the fourth parameter **")"** of the function indicates that we want to end the string with the string ")"

A.2.3 getAllTargetColList() Method

Use to return information about all columns of the target table of an interface, including active and non-active columns. Active columns are those having an active mapping.

This method has the same usage and parameters as the getTargetTable() Method. See Section A.2.59, "getTargetColList() Method" for more details.

A.2.4 getCatalogName() Method

Use to return a catalog name from the topology.

Usage

```
public java.lang.String getCatalogName(
 java.lang.String pLogicalSchemaName,
java.lang.String pLocation)

public java.lang.String getCatalogName(
 java.lang.String pLogicalSchemaName,
java.lang.String pContextCode,
pContextCode, java.lang.String pLocation)

public java.lang.String getCatalogName(
java.lang.String pLocation)

public java.lang.String getCatalogName()
```

Description

Allows you to retrieve the name of a physical data catalog or work catalog, from its logical schema.

If the first syntax is used, the returned catalog name matches the current context.

If the second syntax is used, the returned catalog name is that of the context specified in the pContextCode parameter.

The third syntax returns the name of the data catalog (D) or work catalog (W) for the current logical schema in the current context.

The fourth syntax returns the name of the data catalog (D) for the current logical schema in the current context.

Parameters

Parameter	Type	Description
pLogicalSchemaName	String	Name of the logical schema
pContextCode	String	Code of the enforced context of the schema

Parameter	Type	Description
pLocation	String	The valid values are: ■ W: Returns the work catalog of the physical schema that corresponds to the tuple (context, logical schema) ■ D: Returns the data catalog of the physical schema that corresponds to the tuple (context, logical schema)

Examples

If you have defined the physical schema `Pluton.db_odi.dbo`

Data catalog:	db_odi
Data schema:	dbo
Work catalog:	tempdb
Work schema:	temp_owner

that you have associated with this physical schema: MSSQL_ODI in the context CTX_DEV

The Call To	Returns
<%=odiRef.getCatalogName("MSSQL_ODI", "CTX_DEV", "W")%>	tempdb
<%=odiRef.getCatalogName("MSSQL_ODI", "CTX_DEV", "D")%>	db_odi

A.2.5 getCatalogNameDefaultPSchema() Method

Use to return a catalog name for the default physical schema from the topology.

Usage

```
public java.lang.String getCatalogNameDefaultPSchema(
java.lang.String pLogicalSchemaName,
java.lang.String pLocation)

public java.lang.String getCatalogNameDefaultPSchema(
java.lang.String pLogicalSchemaName,
java.lang.String pContextCode,
java.lang.String pLocation)

public java.lang.String getCatalogNameDefaultPSchema(
java.lang.String pLocation)

public java.lang.String getCatalogNameDefaultPSchema()
```

Description

Allows you to retrieve the name of the **default** physical data catalog or work catalog for the data server to which is associated the physical schema corresponding to the tuple (logical schema, context). If no context is specified, the current context is used. If no logical schema name is specified, then the current logical schema is used. If no pLocation is specified, then the data catalog is returned.

Parameters

Parameter	Type	Description
pLogicalSchemaName	String	Name of the logical schema
pContextCode	String	Code of the enforced context of the schema
pLocation	String	The valid values are: ■ W: Returns the work catalog of the default physical schema associate to the data server to which the physical schema corresponding to the tuple (context, logical schema) is also attached. ■ D: Returns the data catalog of the physical schema corresponding to the tuple (context, logical schema)

Examples

If you have defined the physical schema `Pluton.db_odi.dbo`

Data catalog:	db_odi
Data schema:	dbo
Work catalog:	tempdb
Work schema:	temp_odi
Default Schema	Yes

that you have associated with this physical schema: MSSQL_ODI in the context CTX_DEV, and Pluton.db_doc.doc

Data catalog:	db_doc
Data schema:	doc
Work catalog:	tempdb
Work schema:	temp_doc
Default Schema	No

that you have associated with this physical schema: MSSQL_DOC in the context CTX_DEV.

The Call To	Returns
<%=odiRef.getCatalogNameDefaultPSchema("MSSQL_DOC", "CTX_DEV", "W")%>	tempdb
<%=odiRef.getCatalogNameDefaultPSchema("MSSQL_DOC", "CTX_DEV", "D")%>	db_odi

A.2.6 getCK() Method

Use to return information about a condition.

Usage

```
public java.lang.String getCK(java.lang.String pPropertyName)
```

Description

This method returns information relative to a condition of a datastore during a check procedure. It is accessible from a Check Knowledge Module only if the current task is tagged as "condition".

In an action, this method returns information related to the check constraint currently handled by the DDL command.

Parameters

Parameter	Type	Description
pPropertyName	String	Current string containing the name of the requested property.

The following table lists the different values accepted by pPropertyName:

Parameter Value	Description
ID	Internal number of the check constraint.
COND_ALIAS	Alias of the table used in the SQL statement
COND_NAME	Name of the condition
COND_TYPE	Type of the condition
COND_SQL	SQL statement of the condition
MESS	Error message relative to the check constraint
FULL_NAME	Full name of the check constraint generated with the local object mask.
COND_SQL_DDL	SQL statement of the condition with no table alias.
<flexfield code>	Flexfield value for this check constraint.

Examples

```
The current condition is called: <%=snpRep.getCK("COND_NAME")%>

insert into MY_ERROR_TABLE
select *
from MY_CHECKED_TABLE
where (not (<%=odiRef.getCK("COND_SQL")%>))
```

A.2.7 getColDefaultValue() Method

Use to return the default value of a mapped column.

Usage

```
public java.lang.String getColDefaultValue()
```

Description

Returns the default value of the target column of the mapping.

This method can be used in a mapping expression without the <%%> tags. This method call will insert in the generate code the default value set in the column definition. Depending on the column type, this value should be protected with quotes.

Parameters

None.

Examples

```
The default value of my target column is '+ 'odiRef.getColDefaultValue()'
```

A.2.8 getColList() Method

Use to return properties for each column from a filtered list of columns. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getColList(
java.lang.int pDSIndex,
java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd,
java.lang.String pSelector)
```

Alternative syntaxes:

```
public java.lang.String getColList(
java.lang.int pDSIndex,
java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)

public java.lang.String getColList(
java.lang.int pDSIndex,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pSelector)

public java.lang.String getColList(
java.lang.int pDSIndex,
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of columns and expressions for a given dataset. The columns list depends on the phase during which this method is called.

In IKMs only, In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

The pPattern parameter is interpreted and then repeated for each element of the list (selected according to pSelector parameter) and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends with

pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

In the alternative syntax, any parameters not set are set to an empty string.

Note that this method automatically generates lookups with no specific code required.

Loading (LKM)

All active mapping expressions that are executed in the current source set, as well as all the columns from the current source set used in the mapping, filters and joins expressions executed in the staging area appear in this list. The list is sorted by POS, FILE_POS.

If there is a journalized datastore in the source of the interface, the three journalizing pseudo columns `JRN_FLAG`, `JRN_DATE`, and `JRN_SUBSCRIBER` are added as columns of the journalized source datastore.

Integration (IKM)

All current active mapping expressionsin the current interface appear in the list.

The list contains one element for each column that is loaded in the target table of the current interface. The list is sorted by POS, FILE_POS, except when the target table is temporary. In this case it is not sorted.

If there is a journalized datastore in the source of the interface, and it is located in the staging area, the three journalizing pseudo columns `JRN_FLG`, `JRN_DATE`,and `JRN_SUBSCRIBER` are added as columns of the journalized source datastore.

Check (CKM)

All the columns of the target table (with static or flow control) appear in this list.

To distinguish columns mapped in the current interface, you must use the `MAP` selector.

Actions

All the columns of the table handles by the DDL command appear in this list.

In the case of modified, added or deleted columns, the NEW and OLD selectors are used to retrieve either the new version of the old version of the modified column being processed by the DDL command. The list is sorted by POS, FILE_POS when the table loaded is not temporary.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.
pStart	String	This sequence marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of the attributes usable in a pattern is detailed in the Pattern Attributes List below. Each occurrence of the attributes in the pattern string is replaced by its value. Attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.

Parameter	Type	Description
pEnd	String	This sequence marks the end of the string to generate.
pSelector	String	String that designates a Boolean expression that allows to filter the elements of the initial list with the following format: <SELECTOR> <Operator> <SELECTOR> etc. Parenthesis are authorized. Authorized operators: 1. No: NOT or! 2. Or: OR or || 3. And: AND or && Example: (INS AND UPD) OR TRG The description of valid selectors is provided below.

Pattern Attributes List

The following table lists different parameters values as well as their associated description.

Parameter Value	Description
I_COL	Internal identifier of the column
COL_NAME	Name of the column
COL_HEADING	Header of the column
COL_DESC	Description of the column
POS	Position of the column
LONGC	Column length (Precision)
SCALE	Scale of the column
FILE_POS	Beginning (index) of the column
BYTES	Number of physical bytes in the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column
COL_MANDATORY	Mandatory character of the column. Valid values are: ■ 0: null authorized ■ 1: not null
CHECK_FLOW	Flow control flag of the column. Valid values are: ■ 0: do not check ■ 1: check
CHECK_STAT	Static control flag of the column. Valid values are: ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column
COL_DEC_SEP	Decimal symbol of the column
REC_CODE_LIST	List of the record codes retained in the column

Parameter Value	Description
COL_NULL_IF_ERR	Processing flag of the column. Valid values are: ■ 0: Reject ■ 1: Set to null active trace ■ 2: Set to null inactive trace
DEF_VALUE	Default value of the column
EXPRESSION	Text of the expression executed on the source (expression as typed in the mapping or column name making an expression executed on the staging area).
CX_COL_NAME	Computed name of the column used as a container for the current expression on the staging area
ALIAS_SEP	Separator used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
MANDATORY_CLAUSE	Returns `NOT NULL` is the column is mandatory. Otherwise, returns the null keyword for the technology.
DEFAULT_CLAUSE	Returns `DEFAULT <default value>` if any default value exists. Otherwise, returns and empty string.
JDBC_TYPE	Data Services - JDBC Type of the column returned by the driver.
<flexfield code>	Flexfield value for the current column.

Selectors Description

Parameter Value	Description
INS	■ LKM: Not applicable (*) ■ IKM: Only for mapping expressions marked with insertion ■ CKM: Not applicable
UPD	■ LKM: Not applicable (*) ■ IKM: Only for mapping expressions marked with update ■ CKM: Not applicable
TRG	■ LKM: Not applicable (*) ■ IKM: Only for mapping expressions executed on the target ■ CKM: Not applicable
NULL	■ LKM: Not applicable (*) ■ IKM: All mapping expressions loading not nullable columns ■ CKM: All target columns that do not accept null values

Parameter Value	Description
PK	■ LKM: Not applicable (*) ■ IKM: All mapping expressions loading the primary key columns ■ CKM: All the target columns that are part of the primary key
UK	■ LKM: Not applicable (*) ■ IKM: All the mapping expressions loading the update key column chosen for the current interface ■ CKM: Not applicable
REW	■ LKM: Not applicable (*) ■ IKM: All the mapping expressions loading the columns with read only flag not selected ■ CKM: All the target columns with read only flag not selected
UD1	■ LKM: Not applicable (*) ■ IKM: All mapping expressions loading the columns marked UD1 ■ CKM: Not applicable
UD2	■ LKM: Not applicable (*) ■ IKM: All mapping expressions loading the columns marked UD2 ■ CKM: Not applicable
UD3	■ LKM: Not applicable (*) ■ IKM: All mapping expressions loading the columns marked UD3 ■ CKM: Not applicable
UD4	■ LKM: Not applicable (*) ■ IKM: All mapping expressions loading the columns marked UD4 ■ CKM: Not applicable
UD5	■ LKM: Not applicable (*) ■ IKM: All mapping expressions loading the columns marked UD5 ■ CKM: Not applicable
MAP	■ LKM: Not applicable ■ IKM: Not applicable ■ CKM: Flow control: All columns of the target table loaded with expressions in the current interface Static control: All columns of the target table
SCD_SK	LKM, CKM, IKM: All columns marked SCD Behavior: Surrogate Key in the data model definition.
SCD_NK	LKM, CKM, IKM: All columns marked SCD Behavior: Natural Key in the data model definition.
SCD_UPD	LKM, CKM, IKM: All columns marked SCD Behavior: Overwrite on Change in the data model definition.
SCD_INS	LKM, CKM, IKM: All columns marked SCD Behavior: Add Row on Change in the data model definition.
SCD_FLAG	LKM, CKM, IKM: All columns marked SCD Behavior: Current Record Flag in the data model definition.
SCD_START	LKM, CKM, IKM: All columns marked SCD Behavior: Starting Timestamp in the data model definition.

Parameter Value	Description
SCD_END	LKM, CKM, IKM: All columns marked SCD Behavior: Ending Timestamp in the data model definition.
NEW	Actions: the column added to a table, the new version of the modified column of a table.
OLD	Actions: The column dropped from a table, the old version of the modified column of a table.
WS_INS	SKM: The column is flagged as allowing INSERT using Data Services.
WS_UPD	SKM: The column is flagged as allowing UDATE using Data Services.
WS_SEL	SKM: The column is flagged as allowing SELECT using Data Services.

Note: Using certain selectors in an LKM - indicated in the previous table with an * - is possible but not recommended. Only columns mapped on the source in the interface are returned. As a consequence, the result could be incorrect depending on the interface. For example, for the UK selector, the columns of the key that are not mapped or that are not executed on the source will not be returned with the selector.

Examples

If the CUSTOMER table contains the columns (CUST_ID, CUST_NAME, AGE) and we want to generate the following code:

```
create table CUSTOMER (CUST_ID numeric(10) null,
CUST_NAME varchar(50) null, AGE numeric(3) null)
```

The following code is sufficient:

```
create table CUSTOMER
<%=odiRef.getColList("(", "[COL_NAME] [SOURCE_CRE_DT] null", ", ", ")", "")%>
```

Explanation: the getColList function will be used to generate (CUST_ID numeric(10) null, CUST_NAME varchar(50) null, AGE numeric(3) null). It will start and end with a parenthesis and repeat a pattern (column, data type, and null) separated by commas for each column. Thus,

- the first character **"("** of the function indicates that we want to start the string with the string "("
- the second parameter **"[COL_NAME] [SOURCE_CRE_DT] null"** indicates that we want to repeat this pattern for each column. The keywords [COL_NAME] and [SOURCE_CRE_DT] are references to valid keywords of the table Pattern Attribute List
- the third parameter **", "** indicates that we want to separate the interpreted occurrences of the pattern with the string ", ".
- the fourth parameter **")"** of the function indicates that we want to end the string with the string ")"
- the last parameter **""** indicates that we want to repeat the pattern for each column (with no selection)

A.2.9 getColumn() Method

Use to return information about a specific column handled by an action.

Usage

```
public java.lang.String getColumn(
java.lang.String pPattern,
java.lang.String pSelector)

public java.lang.String getColumn(
java.lang.String pPattern)
```

Description

In an action, returns information on a column being handled by an the action.

Parameters

Parameters	Type	Description
pPattern	String	Pattern of values rendered for the column. The list of the attributes usable in a pattern is detailed in the Pattern Attributes List below. Each occurrence of the attributes in the pattern string is replaced by its value. Attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSelector	String	The Selector may take one of the following value: ■ NEW: returns the new version of the modified column or the new column. ■ OLD: returns the old version of the modified column or the dropped column. If the selector is omitted, it is set to OLD for all *drop* actions. Otherwise, it is set to NEW.

Pattern Attributes List

The following table lists different parameters values as well as their associated description.

Parameter Value	Description
I_COL	Internal identifier of the column
COL_NAME	Name of the column
COL_HEADING	Header of the column
COL_DESC	Description of the column
POS	Position of the column
LONGC	Column length (Precision)
SCALE	Scale of the column
FILE_POS	Beginning (index) of the column
BYTES	Number of physical bytes in the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column

Parameter Value	Description
COL_MANDATORY	Mandatory character of the column. Valid values are: ■ 0: null authorized ■ 1: not null
CHECK_FLOW	Flow control flag of the column. Valid values are: ■ 0: do not check ■ 1: check
CHECK_STAT	Static control flag of the column. Valid values are: ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column
COL_DEC_SEP	Decimal symbol of the column
REC_CODE_LIST	List of the record codes retained in the column
COL_NULL_IF_ERR	Processing flag of the column. Valid values are: ■ 0: Reject ■ 1: Set to null active trace ■ 2: Set to null inactive trace
DEF_VALUE	Default value of the column
EXPRESSION	Text of the expression executed on the source (expression as typed in the mapping or column name making an expression executed on the staging area).
CX_COL_NAME	Computed name of the column used as a container for the current expression on the staging area
ALIAS_SEP	Separator used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
MANDATORY_CLAUSE	Returns `NOT NULL` if the column is mandatory. Otherwise, returns the null keyword for the technology.
DEFAULT_CLAUSE	Returns `DEFAULT <default value>` if any default value exists. Otherwise, returns and empty string.
<flexfield code>	Flexfield value for the current column.

A.2.10 getContext() Method

Use to return information about the current context.

Usage

```
public java.lang.String getContext(java.lang.String pPropertyName)
```

Description

This method returns information about to the current execution context.

Parameters

Parameter	Type	Description
pPropertyName	String	String containing the name of the requested property.

The following table lists the different possible values for pPropertyName.

Parameter Value	Description
ID	Internal ID of the context.
CTX_NAME	Name of the context.
CTX_CODE	Code of the context.
CTX_DEFAULT	Returns 1 for the default context, 0 for the other contexts.
<flexfield code>	Flexfield value for this reference.

Examples

```
Current Context = <%=getContext("CTX_NAME")%>
```

A.2.11 getDataSet() Method

Use to return information about a given dataset of an interface.

Usage

```
public java.lang.String getDataSet(
java.lang.Int pDSIndex,
java.lang.String pPropertyName)
```

Description

Retrieves information about for a given dataset of an interface.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.

The following table lists the different possible values for pPropertyName.

Parameter Value	Description
OPERATOR	Operator that applies to the selected dataset. For the first dataset, an empty value is returned.
NAME	Dataset Name.
HAS_JRN	Returns "1" if the dataset one journalized datastore, "0" otherwise.

Examples

```
<%for (int i=0; i < odiRef.getDataSetCount(); i++){%>
<%=odiRef.getDataSet(i, "Operator")%>
select <%=odiRef.getPop("DISTINCT_ROWS")%>
<%=odiRef.getColList(i,"", "[EXPRESSION] [COL_NAME]", ",\n\t", "", "((INS and
!TRG) and REW)")%>
from<%=odiRef.getFrom(i)%>
where<% if (odiRef.getDataSet(i, "HAS_JRN").equals("1")) { %>
JRN_FLAG <> 'D'<%} else {%>(1=1)<% } %>
<%=odiRef.getJoin(i)%>
<%=odiRef.getFilter(i)%>
<%=odiRef.getJrnFilter(i)%>
<%=odiRef.getGrpBy(i)%>
<%=odiRef.getHaving(i)%>
<%}%>
```

A.2.12 getDataSetCount() Method

Use to return the number of datasets of an interface.

Usage

```
public java.lang.Int getDataSetCount()
```

Description

Returns the number of datasets of an interface.

Parameters

None

Examples

```
<%for (int i=0; i < odiRef.getDataSetCount(); i++){%>
<%=odiRef.getDataSet(i, "Operator")%>
select <%=odiRef.getPop("DISTINCT_ROWS")%>
<%=odiRef.getColList(i,"", "[EXPRESSION] [COL_NAME]", ",\n\t", "", "((INS and
!TRG) and REW)")%>
from<%=odiRef.getFrom(i)%>
where<% if (odiRef.getDataSet(i, "HAS_JRN").equals("1")) { %>
JRN_FLAG <> 'D'<%} else {%>(1=1)<% } %>
<%=odiRef.getJoin(i)%>
<%=odiRef.getFilter(i)%>
<%=odiRef.getJrnFilter(i)%>
<%=odiRef.getGrpBy(i)%>
<%=odiRef.getHaving(i)%>
<%}%>
```

A.2.13 getDataType() Method

Use to return the syntax creating a column of a given datatype.

Usage

```
public java.lang.String getDataType(
java.lang.String pDataTypeName,
java.lang.String pDataTypeLength,
java.lang.String pDataTypePrecision)
```

Description

Returns the creation syntax of the following SQL data types: varchar, numeric or date according to the parameters associated to the source or target technology.

Parameters

Parameters	Type	Description
Parameter	Type	Description
pDataTypeName	String	Name of the data type as listed in the table below
pDataTypeLength	String	Length of the data type
pDataTypePrecision	String	Precision of the data type

The following table lists all possible values for pDataTypeName.

Parameter Value	Description
SRC_VARCHAR	Returns the syntax to the source data type varchar
SRC_NUMERIC	Returns the syntax to the source data type numeric
SRC_DATE	Returns the syntax to the source data type date
DEST_VARCHAR	Returns the syntax to the target data type varchar
DEST_NUMERIC	Returns the syntax to the target data type numeric
DEST_DATE	Returns the syntax to the target data type date

Examples

Given the following syntax for these technologies:

Technology	Varchar	Numeric	Date
Oracle	varchar2(%L)	number(%L,%P)	date
Microsoft SQL Server	varchar(%L)	numeric(%L,%P)	datetime
Microsoft Access	Text(%L)	double	datetime

Here are some examples of call to getDataType:

Call	Oracle	SQL Server	Access
`<%=odiRef.getDataType("DEST_VARCHAR", "10", "")%>`	varchar2(10)	varchar(10)	Text(10)
`<%=odiRef.getDataType("DEST_VARCHAR", "10", "5")%>`	varchar2(10)	varchar(10)	Text(10)

Call	Oracle	SQL Server	Access
`<%=odiRef.getDataType("DEST_NUMERIC", "10", "")%>`	number(10)	numeric(10)	double
`<%=odiRef.getDataType("DEST_NUMERIC", "10", "2")%>`	number(10,2)	numeric(10,2)	double
`<%=odiRef.getDataType("DEST_NUMERIC", "", "")%>`	number	numeric	double
`<%=odiRef.getDataType("DEST_DATE", "", "")%>`	date	datetime	datetime
`<%=odiRef.getDataType("DEST_DATE", "10", "2")%>`	date	datetime	datetime

A.2.14 getFilter() Method

Use to return the entire WHERE clause section generated for the filters of an interface.

Usage

```
public java.lang.String getFilter(java.lang.Int pDSIndex)
```

Description

Returns the SQL filters sequence (on the source while loading, on the staging area while integrating) for a given dataset.

In IKMs only, In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.

None

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

A.2.15 getFilterList() Method

Use to return properties for each filter of an interface. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getFilterList(
```

```
java.lang.Int pDSIndex,
java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getFilterList(
java.lang.Int pDSIndex,
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of occurrences of the SQL filters of a given dataset of an interface.

In IKMs only, In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

The parameter pPattern is interpreted and repeated for each element of the list and separated from its predecessor with parameter pSeparator. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains an element for each filter expression executed on the source or target (depending on the Knowledge Module in use).

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.
pStart	String	This sequence marks the beginning of the string to generate.
pPattern	String	The pattern will be repeated for each occurrence of the list. The list of possible in a list is available in the Pattern Attributes List below. Each attribute occurrence in the pattern string is substituted with its value. Attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter is used to separate a pattern from its predecessor.
pEnd	String	This sequence marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as the associated description.

Parameter Value	Description
ID	Filter internal identifier
EXPRESSION	Text of the filter expression.

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilterList("and ","([EXPRESSION])"," and ","")%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

Explanation: the getFilterList function will be used to generate the filter of the SELECT clause that must begin with "and" and repeats the pattern (expression of each filter) separated with "and" for each filter. Thus

- The first parameter **"and"** of the function indicates that we want to start the string with the string "and"
- the second parameter **"([EXPRESSION])"** indicates that we want to repeat this pattern for each filter. The keywords [EXPRESSION] references a valid keyword of the table Pattern Attribute List
- the third parameter **" and "** indicates that we want to separate each interpreted occurrence of the pattern with the string "and ".
- the fourth parameter **""** of the function indicates that we want to end the string with no specific character.

A.2.16 getFK() Method

Use to return information about a foreign key.

Usage

```
public java.lang.String getFK(java.lang.String pPropertyName)
```

Description

This method returns information relative to the foreign key (or join or reference) of a datastore during a check procedure. It is accessible from a Knowledge Module only if the current task is tagged as a "reference".

In an action, this method returns information related to the foreign key currently handled by the DDL command.

Parameters

Parameter	Type	Description
pPropertyName	String	String containing the name of the requested property.

The following table lists the different possible values for pPropertyName.

Parameter Value	Description
ID	Internal number of the reference constraint.
FK_NAME	Name of the reference constraint.
FK_TYPE	Type of the reference constraint.
FK_ALIAS	Alias of the reference table (only used in case of a complex expression)
PK_ALIAS	Alias of the referenced table (only used in case of a complex expression)
ID_TABLE_PK	Internal number of the referenced table.
PK_I_MOD	Number of the referenced model.
PK_CATALOG	Catalog of the referenced table in the current context.
PK_SCHEMA	Physical schema of the referenced table in the current context.
PK_TABLE_NAME	Name of the referenced table.
COMPLEX_SQL	Complex SQL statement of the join clause (if appropriate).
MESS	Error message of the reference constraint
FULL_NAME	Full name of the foreign key generated with the local object mask.
<flexfield code>	Flexfield value for this reference.

Examples

```
The current reference key of my table is called: <%=odiRef.getFK("FK_NAME")%>. It
references the table <%=odiRef.getFK("PK_TABLE_NAME")%> that is in the schema
<%=odiRef.getFK("PK_SCHEMA")%>
```

A.2.17 getFKColList() Method

Use to return information about the columns of a foreign key.

Usage

```
public java.lang.String getFKColList(java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getFKColList(
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of columns part of a reference constraint (foreign key).

The parameter pPattern in interpreted and repeated for each element of the list, and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains one element for each column of the current foreign key. It is accessible from a Check Knowledge Module only if the current task is tagged as a "reference".

In an action, this method returns the list of the columns of the foreign key handled by the DDL command, ordered by their position in the key.

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameter	Type	Description
Parameter	Type	Description
pStart	String	This parameter marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of possible attributes in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern string is substituted with its value. The attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This parameter marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as the associated description.

Parameter Value	Description
I_COL	Column internal identifier
COL_NAME	Name of the column of the key
COL_HEADING	Header of the column of the key
COL_DESC	Description of the column of the key
POS	Position of the column of the key
LONGC	Length (Precision) of the column of the key
SCALE	Scale of the column of the key
FILE_POS	Beginning (index) of the column
BYTES	Number of physical octets of the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column
COL_MANDATORY	Mandatory character of the column. Valid values are: ■ 0: not authorized ■ 1: not null
CHECK_FLOW	Flow control flag of the column. Valid values are: ■ 0: do not check ■ 1: check

Parameter Value	Description
CHECK_STAT	Static control flag of the column. Valid values are: ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column
COL_DEC_SEP	Decimal symbol for the column
REC_CODE_LIST	List of the record codes for the column
COL_NULL_IF_ERR	Column processing flag. Valid values are: ■ 0: Reject ■ 1: Set active trace to null ■ 2: Set inactive trace to null
DEF_VALUE	Default value of the column
EXPRESSION	Not used
CX_COL_NAME	Not used
ALIAS_SEP	Separator used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
PK_I_COL	Internal identifier of the referenced column
PK_COL_NAME	Name of the referenced key column
PK_COL_HEADING	Header of the referenced key column
PK_COL_DESC	Description of the referenced key column
PK_POS	Position of the referenced column
PK_LONGC	Length of the referenced column
PK_SCALE	Precision of the referenced column
PK_FILE_POS	Beginning (index) of the referenced column
PK_BYTES	Number of physical octets of the referenced column
PK_FILE_END_POS	End of the referenced column (FILE_POS + BYTES)
PK_IND_WRITE	Write right flag of the referenced column
PK_COL_MANDATORY	Mandatory character of the referenced column. Valid values are: ■ 0: null authorized ■ 1: not null
PK_CHECK_FLOW	Flow control flag of the referenced column. Valid values are: ■ 0: do not check ■ 1: check

Parameter Value	Description
PK_CHECK_STAT	Static control flag of the referenced column. Valid values are: ■ 0: do not check ■ 1: check
PK_COL_FORMAT	Logical format of the referenced column
PK_COL_DEC_SEP	Decimal separator for the referenced column
PK_REC_CODE_LIST	List of record codes retained for the referenced column
PK_COL_NULL_IF_ERR	Processing flag of the referenced column. Valid values are: ■ 0: Reject ■ 1: Set active trace to null ■ 2: Set inactive trace to null
PK_DEF_VALUE	Default value of the referenced column
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
<flexfield code>	Flexfield value for the current column of the referencing table.

Examples

If the CUSTOMER table references the CITY table on CUSTOMER.COUNTRY_ID = CITY.ID_COUNT and CUSTOMER.CITY_ID = CITY.ID_CIT

the clause:

```
(CUS.COUNTRY_ID = CITY.ID_COUNT and CUS.CITY_ID = CITY.ID_CIT)
```

can also be written:

```
<%=odiRef.getFKColList("(", "CUS.[COL_NAME] = CITY.[PK_COL_NAME]", " and
", ")" )%>
```

Explanation: the getFKColList function will be used to loop on each column of the foreign key to generate the clause that begins and ends with a parenthesis and that repeats a pattern separated by **and** for each column in the foreign key. Thus

- The first parameter **"("** of the function indicates that we want to begin the string with "("
- The second parameter **"CUS.[COL_NAME] = CITY.[PK_COL_NAME]"** indicates that we want to repeat this pattern for each column of the foreign key. The keywords [COL_NAME] and [PK_COL_NAME] reference valid keywords in the table Pattern Attributes List
- The third parameter**" and "** indicates that we want to separate the occurrences of the pattern with the string " and ".
- The fourth parameter **")"** of the function indicates that we want to end the string with ")".

A.2.18 getFlexFieldValue() Method

Use to return the value of a flexfield.

Usage

```
public java.lang.String getFlexFieldValue(java.lang.String pI_Instance,
java.lang.String pI_Object, java.lang.String pFlexFieldCode)
```

Description

This method returns the value of an Object Instance's Flexfield.

Parameters

Parameter	Type	Description
pI_Instance	String	**Internal Identifier** of the Object Instance, as it appears in the version tab of the object instance window.
pI_Object	String	**Internal Identifier** of the Object type, as it appears in the version tab of the **object** window for the object type.
pPropertyName	String	Flexfield Code which value should be returned.

Examples

```
<%=odiRef.getFlexFieldValue("32001","2400","MY_DATASTORE_FIELD")%>
```

Returns the value of the flexfield MY_DATASTORE_FIELD, for the object instance of type datastore (Internal ID for datastores is 2400), with the internal ID 32001.

A.2.19 getFrom() Method

Use to return the SQL FROM clause in the given context.

Usage

```
public java.lang.String getFrom(java.lang.Int pDSIndex)
```

Description

Allows the retrieval of the SQL string of the **FROM** in the source **SELECT** clause for a given dataset. The **FROM** statement is built from tables and joins (and according to the SQL capabilities of the technologies) that are used in this dataset.

For a technology that supports ISO outer joins and parenthesis, getFrom() could return a string such as:

```
((CUSTOMER as CUS inner join CITY as CIT on (CUS.CITY_ID = CIT.CITY_ID))
left outer join SALES_PERSON as SP on (CUS.SALES_ID = SP.SALE_ID))
```

In IKMs only, In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

If there is a journalized datastore in source of the interface, the source table in the clause is replaced by the data view linked to the journalized source datastore.

If one of the source datastores is a temporary datastore with the Use Temporary Interface as Derived Table (Sub-Select) box selected then a sub-select statement will be generated for this temporary source by the getFrom method.

If partitioning is used on source datastores, this method automatically adds the partitioning clauses when returning the object names.

Note that this method automatically generates lookups with no specific code required.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

A.2.20 getGrpBy() Method

Use to return the entire SQL GROUP BY clause in the given context.

Usage

```
public java.lang.String getGrpBy(java.lang.Int pDSIndex)
```

Description

Allows you to retrieve the SQL GROUP BY string (on the "source" during the loading phase, on the staging area during the integration phase) for a given dataset. This statement is automatically computed from the aggregation transformations detected in the mapping expressions.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
```

```
<%=odiRef.getFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

A.2.21 getGrpByList() Method

Use to return properties for each GROUP BY clause for a given dataset in an interface. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getGrpByList(
java.lang.Int pDSIndex,
java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getGrpByList(
java.lang.Int pDSIndex,
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of occurrences of SQL GROUP BY for a given dataset of an interface.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

The pPattern parameter is interpreted, then repeated for each element of the list and separated from its predecessor with the pSeparator parameter. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains an element for each GROUP BY statement on the source or target (according to the Knowledge Module that used it).

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.
pStart	String	This parameter marks the beginning of the string to generate.

Parameter	Type	Description
pPattern	String	The pattern is repeated for each occurrence in the list. The list of possible attributes in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern string is substituted with its value. The attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter is used to separate each pattern from its predecessor.
pEnd	String	This parameter marks the end of the string to be generated.

Pattern Attributes List

The following table lists the different values of the parameters as well as their associated description.

Parameter Value	Description
ID	Internal identifier of the clause
EXPRESSION	Text of the grouping statement

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getGrpByList("group by ", "[EXPRESSION]", " , ", "")%>
<%=odiRef.getHaving()%>
```

Explanation: the getGrpByList function will be used to generate the **group by** clause of the **select** order that must start with "group by" and that repeats a pattern (each grouping expression) separated by commas for each expression.

- The first parameter **"group by"** of the function indicates that we want to start the string with "group by"
- The second parameter **"[EXPRESSION]"** indicates that we want to repeat this pattern for each group by expression. The keyword [EXPRESSION] references a valid keyword of the table Pattern Attributes List
- The third parameter **", "** indicates that we want to separate the interpreted occurrences of the pattern with a comma.
- The fourth parameter **""** of the function indicates that we want to end the string with no specific character

A.2.22 getHaving() Method

Use to return the entire SQL HAVING clause in the given context.

Usage

```
public java.lang.String getHaving(java.lang.Int pDSIndex)
```

Description

Allows the retrieval of the SQL statement HAVING (on the source during loading, on the staging area during integration) for a given dataset. This statement is automatically computed from the filter expressions containing detected aggregation functions.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.

Examples

```
insert into <%=odiRef.getTable(
"L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
 <%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

A.2.23 getHavingList() Method

Use to return properties for each HAVING clause of an interface. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getHavingList(
java.lang.Int pDSIndex,
java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getHavingList(
java.lang.Int pDSIndex,
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of the occurrences of SQL HAVING of a given dataset in an interface.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

Note: The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

The parameter pPattern is interpreted and repeated for each element of the list, and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains one element for each HAVING expression to execute on the source or target (depends on the Knowledge module that uses it).

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameters	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.
pStart	String	This parameter marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of authorized attributes in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern string is substituted with its value. The attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This parameter marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as the associated description.

Parameter Value	Description
Parameter value	Description
ID	Internal identifier of the clause
EXPRESSION	Text of the having expression

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getGrpByList("group by ", "[EXPRESSION]", " , ", "")%>
<%=odiRef.getHavingList("having ", "([EXPRESSION])", " and ", "")%>
```

Explanation: The getHavingList function will be used to generate the having clause of the select order that must start with "having" and that repeats a pattern (each aggregated filtered expression) separated by "and" for each expression.

- The first parameter **"having "** of the function indicates that we want to start the string with "having"
- The second parameter **"([EXPRESSION])"** indicates that we want to repeat this pattern for each aggregated filter. The keyword [EXPRESSION] references a valid keyword of the table Pattern Attributes List
- The third parameter**" and "** indicates that we want to separate each interpreted occurrence of the pattern with the string " and ".
- The fourth parameter **""** of the function indicates that we want to end the string with no specific character

A.2.24 getIndex() Method

Use to return information about a specific index handled by an action.

Usage

```
public java.lang.String getIndex(java.lang.String pPropertyName)
```

Description

In an action, this method returns information related to the index currently handled by the DDL command.

Parameters

Parameter	Type	Description
pPropertyName	String	String containing the name of the requested property.

The following table lists the different possible values for pPropertyName.

Parameter Value	Description
ID	Internal number of the index.
KEY_NAME	Name of the index
FULL_NAME	Full name of the index generated with the local object mask.
<flexfield code>	Value of the flexfield for this index.

A.2.25 getIndexColList() Method

Use to return information about the columns of an index handled by an action.

Usage

```
public java.lang.String getIndexColList(java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Description

In an action, this method returns the list of the columns of the index handled by the DDL command, ordered by their position in the index.

The pPattern parameter is interpreted and then repeated for each element of the list. It is separated from its predecessor by the pSeparator parameter. The generated string starts with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains an element for each column of the current index.

Parameters

Parameters	Type	Description
pStart	String	This sequence marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of attributes that can be used in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern sequence is replaced with its value. The attributes must be between brackets. ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This sequence marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as their associated description.

Parameter Value	Description
I_COL	Column internal identifier
COL_NAME	Name of the index column
COL_HEADING	Header of the index column
COL_DESC	Column description
POS	Position of the column
LONGC	Length (Precision) of the column
SCALE	Scale of the column
FILE_POS	Beginning position of the column (fixed file)
BYTES	Number of physical bytes of the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column
COL_MANDATORY	Mandatory character of the column. Valid values are: ■ 0: null authorized ■ 1: non null
CHECK_FLOW	Flow control flag for of the column. Valid values are: ■ 0: do not check ■ 1: check

Parameter Value	Description
CHECK_STAT	Static control flag of the column. Valid values are: ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column
COL_DEC_SEP	Decimal symbol for the column
REC_CODE_LIST	List of the record codes retained for the column
COL_NULL_IF_ERR	Processing flag for the column. Valid values are: ■ 0: Reject ■ 1: Set active trace to null ■ 2: Set inactive trace to null
DEF_VALUE	Default value for the column
EXPRESSION	Not used
CX_COL_NAME	Not used
ALIAS_SEP	Grouping symbol used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
<flexfield code>	Flexfield value for the current column.

A.2.26 getInfo() Method

Use to return information about the current task.

Usage

```
public java.lang.String getInfo(java.lang.String pPropertyName)
```

Description

This method returns information about the current task. The list of available information is described in the pPropertyName values table.

Parameters

Parameter	Type	Description
pPropertyName	String	String containing the name of the requested property.

The following table lists the different values possible for pPropertyName:

Parameter Value	Description
I_SRC_SET	Internal identifier of the current Source Set if the task belongs to a Loading Knowledge Module
SRC_SET_NAME	Name of the current Source Set if the task belongs to a Loading Knowledge Module
COLL_NAME	Name of the current loading resource (C$) if the task belongs to a Loading Knowledge Module
INT_NAME	Name of the current integration resource (I$) if the task belongs to a string Loading, Integration or Check Knowledge Module.
ERR_NAME	Name of the current error resource (E$) if the task is part of a Loading, Integration or Check Knowledge Module
TARG_NAME	Name of the target resource if the task is part of a Loading, Integration or Check Knowledge Module
SRC_CATALOG	Name of the data catalog in the source environment
SRC_SCHEMA	Name of the data schema in the source environment
SRC_WORK_CATALOG	Name of the work catalog in the source environment
SRC_WORK_SCHEMA	Name of the work schema in the source environment
DEST_CATALOG	Name of the data catalog in the target environment
DEST_SCHEMA	Name of the data schema in the target environment
DEST_WORK_CATALOG	Name of the work catalog in the target environment
DEST_WORK_SCHEMA	Name of the work schema in the target environment
SRC_TECHNO_NAME	Name of the source technology
SRC_CON_NAME	Name of the source connection
SRC_DSERV_NAME	Name of the data server of the source machine
SRC_CONNECT_TYPE	Connection type of the source machine
SRC_IND_JNDI	JNDI URL flag
SRC_JAVA_DRIVER	Name of the JDBC driver of the source connection
SRC_JAVA_URL	JDBC URL of the source connection
SRC_JNDI_AUTHENT	JNDI authentication type
SRC_JNDI_PROTO	JNDI source protocol
SRC_JNDI_FACTORY	JNDI source Factory
SRC_JNDI_URL	Source JNDI URL
SRC_JNDI_RESSOURCE	Accessed source JNDI resource
SRC_JNDI_USER	User name for JNDI authentication on the source.
SRC_JNDI_ENCODED_PASS	Encrypted password for JNDI authentication on the source.
SRC_USER_NAME	User name of the source connection
SRC_ENCODED_PASS	Encrypted password of the source connection
SRC_FETCH_ARRAY	Size of the source array fetch
SRC_BATCH_UPDATE	Size of the source batch update
SRC_EXE_CHANNEL	Execution canal of the source connection

Parameter Value	Description
SRC_COL_ALIAS_WORD	Term used to separated the columns from their aliases for the source technology
SRC_TAB_ALIAS_WORD	Term used to separated the tables from their aliases for the source technology
SRC_DATE_FCT	Function returning the current date for the source technology
SRC_DDL_NULL	Returns the definition used for the keyword NULL during the creation of a table on the source
SRC_MAX_COL_NAME_LEN	Maximum number of characters for the column name on the source technology
SRC_MAX_TAB_NAME_LEN	Maximum number of characters for the table name on the source technology
SRC_REM_OBJ_PATTERN	Substitution model for a remote object on the source technology.
SRC_LOC_OBJ_PATTERN	Substitution model for a local object name on the source technology.
DEST_TECHNO_NAME	Name of the target technology
DEST_CON_NAME	Name of the target connection
DEST_DSERV_NAME	Name of the data server of the target machine
DEST_CONNECT_TYPE	Connection type of the target machine
DEST_IND_JNDI	Target JNDI URL flag
DEST_JAVA_DRIVER	Name of the JDBC driver of the target connection
DEST_JAVA_URL	JDBC URL of the target connection
DEST_JNDI_AUTHENT	JNDI authentication type of the target
DEST_JNDI_PROTO	JNDI target protocol
DEST_JNDI_FACTORY	JNDI target Factory
DEST_JNDI_URL	JNDI URL of the target
DEST_JNDI_RESSOURCE	Target JNDI resource that is accessed
DEST_JNDI_USER	User name for JNDI authentication on the target.
DEST_JNDI_ENCODED_PASS	Encrypted password for JNDI authentication on the target.
DEST_USER_NAME	Name of the user for the target connection
DEST_ENCODED_PASS	Encrypted password for the target connection
DEST_FETCH_ARRAY	Size of the target array fetch
DEST_BATCH_UPDATE	Size of the target batch update
DEST_EXE_CHANNEL	Execution canal of the target connection
DEST_COL_ALIAS_WORD	Term used to separate the columns from their aliases on the target technology
DEST_TAB_ALIAS_WORD	Term used to separate the tables from their aliases on the target technology
DEST_DATE_FCT	Function returning the current date on the target technology
DEST_DDL_NULL	Function returning the definition used for the keyword NULL during the creation on a table on the target
DEST_MAX_COL_NAME_LEN	Maximum number of characters of the column in the target technology
DEST_MAX_TAB_NAME_LEN	Maximum number of characters of the table name on the target technology
DEST_REM_OBJ_PATTERN	Substitution model for a remote object on the target technology

Parameter Value	Description
DEST_LOC_OBJ_PATTERN	Substitution model for a local object name on the target technology
CT_ERR_TYPE	Error type (F: Flow, S: Static). Applies only in the case of a Check Knowledge Module
CT_ERR_ID	Error identifier (Table # for a static control or interface number for flow control. Applies only in the case of a Check Knowledge Module
CT_ORIGIN	Name that identifies the origin of an error (Name of a table for static control, or name of an interface prefixed with the project code). Applies only in the case of a Check Knowledge Module
JRN_NAME	Name of the journalized datastore.
JRN_VIEW	Name of the view linked to the journalized datastore.
JRN_DATA_VIEW	Name of the data view linked to the journalized datastore.
JRN_TRIGGER	Name of the trigger linked to the journalized datastore.
JRN_ITRIGGER	Name of the Insert trigger linked to the journalized datastore.
JRN_UTRIGGER	Name of the Update trigger linked to the journalized datastore.
JRN_DTRIGGER	Name of the Delete trigger linked to the journalized datastore.
SUBSCRIBER_TABLE	Name of the datastore containing the subscribers list.
CDC_SET_TABLE	Full name of the table containing list of CDC sets.
CDC_TABLE_TABLE	Full name of the table containing the list of tables journalized through CDC sets.
CDC_SUBS_TABLE	Full name of the table containing the list of subscribers to CDC sets.
CDC_OBJECTS_TABLE	Full name of the table containing the journalizing parameters and objects.
SRC_DEF_CATALOG	Default catalog for the source data server.
SRC_DEF_SCHEMA	Default schema for the source data server.
SRC_DEFW_CATALOG	Default work catalog for the source data server.
SRC_DEFW_SCHEMA	Default work schema for the source data server.
DEST_DEF_CATALOG	Default catalog for the target data server.
DEST_DEF_SCHEMA	Default schema for the target data server.
DEST_DEFW_CATALOG	Default work catalog for the target data server.
DEST_DEFW_SCHEMA	Default work schema for the target data server.
SRC_LSCHEMA_NAME	Source logical schema name.
DEST_LSCHEMA_NAME	Target logical schema name.
SRC_I_CONNECT	Internal ID of the source data server.
SRC_I_PSCHEMA	Internal ID of the source physical schema.
SRC_I_LSCHEMA	Internal ID of the source logical schema.
SRC_I_TECHNO	Internal ID of the source technology.
DEST_I_CONNECT	Internal ID of the target data server.
DEST_I_PSCHEMA	Internal ID of the target physical schema.
DEST_I_LSCHEMA	Internal ID of the target logical schema.
DEST_I_TECHNO	Internal ID of the target technology.

Examples

```
The current source condition is: <%=odiRef.getInfo("SRC_CON_NAME")%>  on server:
<%=odiRef.getInfo("SRC_DSERV_NAME")%>
```

A.2.27 getJDBCConnection() Method

Use to return the source or target JDBC connection.

Usage

```
java.sql.Connection getJDBCConnection(
java.lang.String pPropertyName)
```

Description

This method returns the source or target JDBC connection for the current task.

> **Note:** This method does not return a string, but a JDBC connection object. This object may be used in your Java code within the task.

Parameters

Parameter	Type	Description
pPropertyName	String	Name of connection to be returned.

The following table lists the different values possible for pPropertyName:

Parameter Value	Description
SRC	Source connection for the current task.
DEST	Target connection for the current task.
WORKREP	Work Repository connection.

Examples

Gets the source connection and creates a statement for this connection.

```
java.sql.Connection sourceConnection = odiRef.getJDBCConnection("SRC");
java.sql.Statement s = sourceConnection.createStatement();
```

A.2.28 getJDBCConnectionFromLSchema() Method

Use to return a JDBC connection for a given logical schema.

Usage

```
public java.lang.String getJDBCConnectionFromLSchema(
java.lang.String pLogicalSchemaName,
java.lang.String pContextName)

public java.lang.String getJDBCConnectionFromLSchema(
java.lang.String pLogicalSchemaName)
```

Description

Returns a JDBC connection for a given logical schema. The pLogicalSchemaName identifies the logical schema.

The first syntax resolves the logical schema in the context provided in the pContextName parameter.

The second syntax resolves the logical schema in the current context.

Parameters

Parameter	Type	Description
pLogicalSchemaName	String	Name of the forced logical schema of the object.
pContextName	String	Forced context of the object

Note: This method does not return a string, but a JDBC connection object. This object may be used in your Java code within the task.

A.2.29 getJoin() Method

Use to return the entire WHERE clause section generated for the joins of an interface.

Usage

```
public java.lang.String getJoin(java.lang.Int pDSIndex)
```

Description

Retrieves the SQL join string (on the source during the loading, on the staging area during the integration) for a given dataset of an interface.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

Note: The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoin()%>
<%=odiRef.getFilter()%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

A.2.30 getJoinList() Method

Use to return properties for each join of an interface. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getJoinList(
java.lang.Int pDSIndex,
java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getJoinList(
java.lang.Int pDSIndex,
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of the occurrences of the SQL joins in a given dataset of an interface for the WHERE clause.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

The pPattern parameter is interpreted and then repeated for each element in the list and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends up with pEnd.

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.
pStart	String	This parameter marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of authorized attributes in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern string is substituted with its value. The attributes must be between brackets ([and]) Example My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.

Parameter	Type	Description
pEnd	String	This parameter marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as the associated description.

Parameter Value	Description
ID	Internal identifier of the join
EXPRESSION	Text of the join expression

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getFrom()%>
where (1=1)
<%=odiRef.getJoinList("and ","([EXPRESSION])"," and ","")%>
<%=odiRef.getFilterList("and ","([EXPRESSION])"," and ","")%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

Explanation: the getJoinList function will be used to generate join expressions to put in the WHERE part of the SELECT statement that must start with "and" and that repeats a pattern (the expression of each join) separated by " and " for each join. Thus:

- The first parameter **"and"** of the function indicates that we want to start the string with "and"
- The second parameter **"([EXPRESSION])"** indicates that we want to repeat this pattern for each join. The keyword [EXPRESSION] references a valid keyword of the table Pattern Attributes List
- The third parameter **" and "** indicates that we want to separate each interpreted occurrence of the pattern with " and " (note the spaces before and after "and")
- The fourth parameter **""** of the function indicates that we want to end the string with no specific character

A.2.31 getJrnFilter() Method

Use to return the journalizing filter of an interface.

Usage

```
public java.lang.String getJrnFilter(java.lang.Int pDSIndex)
```

Description

Returns the SQL Journalizing filter for a given dataset in the current interface. If the journalized table in the source, this method can be used during the loading phase. If the journalized table in the staging area, this method can be used while integrating.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

Note: The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.

Examples

```
<%=odiRef.getJrnFilter()%>
```

A.2.32 getJrnInfo() Method

Use to return journalizing information about a datastore.

Usage

```
public java.lang.String getJrnInfo(java.lang.String pPropertyName)
```

Description

Returns information about a datastore's journalizing for a JKM while journalizing a model/datastore, or for a LKM/IKM in an interface.

Parameters

Parameter	Type	Description
pPropertyName	String	String containing the name of the requested property.

The following table lists the different values possible for pPropertyName:

Parameter Value	Description
FULL_TABLE_NAME	Full name of the journalized datastore.
JRN_FULL_NAME	Full name of the journal datastore.
JRN_FULL_VIEW	Full name of the view linked to the journalized datastore.
JRN_FULL_DATA_VIEW	Full name of the data view linked to the journalized datastore.
JRN_FULL_TRIGGER	Full name of the trigger linked to the journalized datastore.
JRN_FULL_ITRIGGER	Full name of the Insert trigger linked to the journalized datastore.
JRN_FULL_UTRIGGER	Full name of the Update trigger linked to the journalized datastore.
JRN_FULL_DTRIGGER	Full name of the Delete trigger linked to the journalized datastore.
SNP_JRN_SUBSCRIBER	Name of the subscriber table in the work schema.
JRN_NAME	Name of the journalized datastore.
JRN_VIEW	Name of the view linked to the journalized datastore.
JRN_DATA_VIEW	Name of the data view linked to the journalized datastore.

Parameter Value	Description
JRN_TRIGGER	Name of the trigger linked to the journalized datastore.
JRN_ITRIGGER	Name of the Insert trigger linked to the journalized datastore.
JRN_UTRIGGER	Name of the Update trigger linked to the journalized datastore.
JRN_DTRIGGER	Name of the Delete trigger linked to the journalized datastore.
JRN_SUBSCRIBER	Name of the subscriber.
JRN_COD_MOD	Code of the journalized data model.
JRN_METHOD	Journalizing Mode (consistent or simple).
CDC_SET_TABLE	Full name of the table containing list of CDC sets.
CDC_TABLE_TABLE	Full name of the table containing the list of tables journalized through CDC sets.
CDC_SUBS_TABLE	Full name of the table containing the list of subscribers to CDC sets.
CDC_OBJECTS_TABLE	Full name of the table containing the journalizing parameters and objects.

Examples

```
The table being journalized is <%=odiRef.getJrnInfo("FULL_TABLE_NAME")%>
```

A.2.33 getLoadPlanInstance() Method

Use to return the Load Plan instance information.

Usage

```
public java.lang.String getLoadPlanInstance (java.lang.String pPropertyName)
```

Description

This method returns the current execution instance information for a Load Plan.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the possible values for pPropertyName:

Parameter Value	Description
BATCH_ID	Load Plan instance identifier (also Instance ID). Every time a Load Plan is started, a new Load Plan instance with a unique identifier is created
RESTART_ATTEMPTS	Number of execution attempts of this Load Plan instance (also Run #). It starts at 1 when the Load Plan instance is first executed, and is incremented each time the Load Plan instance is restarted.
LOAD_PLAN_NAME	Name of the Load Plan
START_DATE	Starting date and time of the current Load Plan instance run

Examples

The current Load Plan `<%=odiRef.getLoadPlanInstance("LOAD_PLAN_NAME")%>` started execution at `<%=odiRef.getLoadPlanInstance("START_DATE")%>`

A.2.34 getModel() Method

Use to return information about a model.

Usage

```
public java.lang.String getModel(java.lang.String pPropertyName)
```

Description

This method returns information on the current data model during the processing of a personalized reverse engineering. The list of available data is described in the pPropertyName values table.

> **Note:** This method may be used on the source connection (data server being reverse-engineered) as well as on the target connection (repository). On the target connection, only the properties independent from the context can be specified (for example, the schema and catalog names cannot be used).

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the possible values for pPropertyName:

Parameter Value	Description
ID	Internal identifier of the current model
MOD_NAME	Name of the current model
LSCHEMA_NAME	Name of the logical schema of the current model
MOD_TEXT	Description of the current model
REV_TYPE	Reverse engineering type: S for standard reverse, C for customize
REV_UPDATE	Update flag of the model
REV_INSERT	Insert flag for the model
REV_OBJ_PATT	Mask for the objects to reverse.

Parameter Value	Description
REV_OBJ_TYPE	List of object types to reverse-engineer for this model. This is a semicolon separated list of object types codes. Valid codes are: ■ T: Table ■ V: View ■ Q: Queue ■ SY: System table ■ AT: Table alias ■ SY: Synonym
TECH_INT_NAME	Internal name of the technology of the current model.
LAGENT_NAME	Name of the logical execution agent for the reverse engineering.
REV_CONTEXT	Execution context of the reverse
REV_ALIAS_LTRIM	Characters to be suppressed for the alias generation
CKM	Check Knowledge Module
RKM	Reverse-engineering Knowledge Module
SCHEMA_NAME	Physical Name of the data schema in the current reverse context
WSCHEMA_NAME	Physical Name of the work schema in the current reverse context
CATALOG_NAME	Physical Name of the data catalog in the current reverse context
WCATALOG_NAME	Physical Name of the work catalog in the current reverse context
<flexfield code>	Value of the flexfield for the current model.

Examples

Retrieve the list of tables that are part of the mask of objects to reverse:

```
select TABLE_NAME,
         RES_NAME,
     replace(TABLE_NAME, '<%=odiRef.getModel("REV_ALIAS_LTRIM")%>' , '')
          ALIAS,
          TABLE_DESC
from MY_TABLES
where
TABLE_NAME like '<%=odiRef.getModel("REV_OBJ_PATT")%>'
```

A.2.35 getNbInsert(), getNbUpdate(), getNbDelete(), getNbErrors() and getNbRows() Methods

Use to get the number of inserted, updated, deleted or erroneous rows for the current task.

Usage

```
public java.lang.Long getNbInsert()

public java.lang.Long getNbUpdate()

public java.lang.Long getNbDelete()

public java.lang.Long getNbErrors()
```

```
public java.lang.Long getNbRows()
```

Description

These methods get for the current task the values for:

- the number of rows inserted (getNbInsert)
- the number of rows updated (getNbUpdate)
- the number of rows deleted (getNbDelete)
- the number of rows in error (getNbErrors)
- total number of rows handled during this task (getNbRows)

These numbers can be set independently from the real number of lines processed using the setNbInsert(), setNbUpdate(), setNbDelete(), setNbErrors() and setNbRows() Methods.

Examples

In the Jython example below, we set the number of inserted rows to the constant value of 50, and copy this value in the number of errors.

```
InsertNumber=50

odiRef.setNbInsert(InsertNumber)

odiRef.setNbErrors(odiRef.getNbInsert())
```

A.2.36 getNewColComment() Method

Use to return the new comment for a specific column handled by an action.

Usage

```
public java.lang.String getNewColComment()
```

Description

In an action, this method returns the new comment for the column being handled by the DDL command, in a *Modify column comment action*.

A.2.37 getNewTableComment() Method

Use to return the new comment for a specific table handled by an action.

Usage

```
public java.lang.String getNewTableComment()
```

Description

In an action, this method returns the new comment for the table being handled by the DDL command, in a *Modify table comment action*.

A.2.38 getNotNullCol() Method

Use to return information about a column that is checked for not null.

Usage

```
public java.lang.String getNotNullCol(java.lang.String pPropertyName)
```

Description

This method returns information relative to a not null column of a datastore during a check procedure. It is accessible from a Check Knowledge Module if the current task is tagged as "mandatory".

Parameters

Parameter	Type	Description
Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the different possible values for pPropertyName:

Parameter Value	Description
ID	Internal identifier for the current column.
COL_NAME	Name of the Not null column.
MESS	Standard error message.
<flexfield code>	Flexfield value for the current not null column.

Examples

```
insert into...
select *
from ...
<%=odiRef.getNotNullCol("COL_NAME")%> is null
```

A.2.39 getObjectName() Method

Use to return the fully qualified named of an object.

Usage

```
public java.lang.String getObjectName(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLocation)

public java.lang.String getObjectName(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLogicalSchemaName,
java.lang.String pLocation)

public java.lang.String getObjectName(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLogicalSchemaName,
java.lang.String pContextName,
java.lang.String pLocation)
```

```
public java.lang.String getObjectName(
java.lang.String pObjectName,
java.lang.String pLocation)

public java.lang.String getObjectName(
java.lang.String pObjectName)

public java.lang.String getObjectName(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLogicalSchemaName,
java.lang.String pContextName,
java.lang.String pLocation,
java.lang.String pPartitionType,
java.lang.String pPartitionName)
```

Description

Returns the fully qualified name of a physical object, including its catalog and schema. The pMode parameter indicates the substitution mask to use.

> **Note:** The getObjectName methods truncates automatically object names to the maximum object length allowed for the technology. In versions before ODI 11g, object names were not trucated. To prevent object names truncation and reproduce the 10g behavior, add in the properties tab of the data server a property called *OBJECT_NAME_LENGTH_CHECK_OLD* and set its value to `true`.

The first syntax builds the object name according to the current logical schema in the current context.

The second syntax builds the name of the object according to the logical schema indicated in the pLogicalSchemaName parameter in the current context.

The third syntax builds the name from the logical schema and the context indicated in the pLogicalSchemaName and pContextName parameters.

The fourth syntax builds the object name according to the current logical schema in the current context, with the local object mask (pMode = "L").

The fifth syntax is equivalent to the fourth with pLocation = "D".

The last syntax is equivalent to the third syntax but qualifies the object name specifically on a given partition, using the pPartitionType and pPartitionName parameters.

Parameters

Parameter	Type	Description
pMode	String	"L" use the local object mask to build the complete path of the object. "R" use the remote object mask to build the complete path of the object. Note: When using the remote object mask, getObjectName always resolved the object name using the default physical schema of the remote server.

Parameter	Type	Description
pObjectName	String	Every string that represents a valid resource name (table or file). This object name may be prefixed by a prefix code that will be replaced at run-time by the appropriate temporary object prefix defined for the physical schema.
pLogicalSchemaName	String	Name of the forced logical schema of the object.
pContextName	String	Forced context of the object
pLocation	String	The valid values are: ■ W: Returns the complete name of the object in the physical catalog and the "work" physical schema that corresponds to the specified tuple (context, logical schema) ■ D: Returns the complete name of the object in the physical catalog and the data physical schema that corresponds to the specified tuple (context, logical schema)
pPartitionType	String	Specify whether to qualify the object name for a specific partition or sub-partition. The valid values are: ■ P: Qualify object for the partition provided in pPartitionName ■ S: Qualify object for the sub-partition provided in pPartitionName
pPartitionName	String	Name of the partition of sub-partition to qualify the object name.

Prefixes

It is possible to prefix the resource name specified in the pObjectName parameter by a prefix code to generate a Oracle Data Integrator temporary object name (Error or Integration table, journalizing trigger, etc.).

The list of prefixes are given in the table below.

Prefix	Description
Prefix	Description
%INT_PRF	Prefix for integration tables (default value is "I$_").
%COL_PRF	Prefix for Loading tables (default value is "C$_").
%ERR_PRF	Prefix for error tables (default value is "E$_").
%JRN_PRF_TAB	Prefix for journalizing tables (default value is "J$_").
%INT_PRF_VIE	Prefix for journalizing view (default value is "JV$_").
%JRN_PRF_TRG	Prefix for journalizing triggers (default value is "T$_").
%IDX_PRF	Prefix for temporary indexes (default value is "IX$_").

Note: Temporary objects are usually created in the work physical schema. Therefore, pLocation should be set to "W" when using a prefix to create or access a temporary object.

Examples

You have defined a physical schema as shown below.

Data catalog:	db_odi
Data schema:	dbo

Work catalog:	tempdb
Work schema:	temp_owner

You have associated this physical schema to the logical schema MSSQL_ODI in the context CTX_DEV.

A Call To	Returns
`<%=odiRef.getObjectName("L", "EMP", "MSSQL_ODI", "CTX_DEV", "W")%>`	tempdb.temp_owner.EMP
`<%=odiRef.getObjectName("L", "EMP", "MSSQL_ODI", "CTX_DEV", "D")%>`	db_odi.dbo.EMP
`<%=odiRef.getObjectName("R", "%ERR_PRFEMP", "MSSQL_ODI", "CTX_DEV", "W")%>`	MyServer.tempdb.temp_owner.E$_EMP
`<%=odiRef.getObjectName("R", "EMP", "MSSQL_ODI", "CTX_DEV", "D")%>`	MyServer.db_odi.dbo.EMP

A.2.40 getObjectNameDefaultPSchema() Method

Use to return the fully qualified named of an object in the default physical schema for the data server.

Usage

```
public java.lang.String getObjectNameDefaultPSchema(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLocation)

public java.lang.String getObjectNameDefaultPSchema(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLogicalSchemaName,
java.lang.String pLocation)

public java.lang.String getObjectNameDefaultPSchema(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLogicalSchemaName,
java.lang.String pContextName,
java.lang.String pLocation)

public java.lang.String getObjectNameDefaultPSchema(
java.lang.String pObjectName,
java.lang.String pLocation)

public java.lang.String getObjectNameDefaultPSchema(
java.lang.String pObjectName)

public java.lang.String getObjectNameDefaultPSchema(
java.lang.String pMode,
java.lang.String pObjectName,
java.lang.String pLogicalSchemaName,
java.lang.String pContextName,
java.lang.String pLocation,
java.lang.String pPartitionType,
java.lang.String pPartitionName)
```

Description

The method is similar to the getObjectName method. However, the object name is computed for the default physical schema of the data server to which the physical schema is attached. In getObjectName, the object name is computed for the physical schema itself.

For more information, see "getObjectName() Method".

A.2.41 getOption() Method

Use to return the value of a KM or procedure option.

Usage

```
public java.lang.String getOption(java.lang.String pOptionName)
public java.lang.String getUserExit(java.lang.String pOptionName)
```

Description

Returns the value of a KM or procedure option.

The getUserExit syntax is deprecated and is only kept for compatibility reasons.

Parameters

Parameter	Type	Description
pOptionName	String	String that contains the name of the requested option.

Examples

```
The value of my MY_OPTION_1 option is <%=odiRef.getOption("MY_OPTION_1")%>
```

A.2.42 getPackage() Method

Use to return information about the current package.

Usage

```
public java.lang.String getPackage(java.lang.String pPropertyName)
```

Description

This method returns information about the current package. The list of available properties is described in the pPropertyName values table.

Parameters

Parameters	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the different possible values for pPropertyName:

Parameter Value	Description
I_PACKAGE	Internal ID of the package

Parameter Value	Description
PACKAGE_NAME	Name of the package
<flexfield code>	Value of the flexfield for this package.

Examples

```
Package <%=odiRef.getPackage("PACKAGE_NAME")%> is running.
```

A.2.43 getParentLoadPlanStepInstance() Method

Use to return the parent Load Plan step instance of this session.

Usage

```
public java.lang.String getParentLoadPlanStepInstance(java.lang.String
pPropertyName)
```

Description

This method returns the step execution instance information of the parent of the current step for a Load Plan instance. It will return an empty string if the parent step is the root step.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the different possible values for pPropertyName.

Parameter Value	Description
BATCH_ID	Load Plan instance identifier (also Instance ID). Every time a Load Plan is started, a new Load Plan instance with a unique identifier is created.
RESTART_ATTEMPTS	Number of execution attempts of this Load Plan parent step instance. It starts at 1 when the Load Plan parent step instance is first started, and is incremented each time the Load Plan parent step instance is restarted.
STEP_NAME	Name of the Load Plan parent step
STEP_TYPE	Type of the Load Plan parent step
START_DATE	Starting date and time of the parent step instance of the current step of the current Load Plan instance run.

Examples

Step `<%=odiRef.getParentLoadPlanStepInstance("STEP_NAME")%>` has been executed `<%=odiRef.getParentLoadPlanStepInstance("RESTART_ATTEMPTS")%>` times

A.2.44 getPK() Method

Use to return information about a primary key.

Usage

```
public java.lang.String getPK(java.lang.String pPropertyName)
```

Description

This method returns information relative to the primary key of a datastore during a check procedure.

In an action, this method returns information related to the primary key currently handled by the DDL command.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the different possible values for pPropertyName.

Parameter Value	Description
ID	Internal number of the PK constraint.
KEY_NAME	Name of the primary key
MESS	Error message relative to the primary key constraint.
FULL_NAME	Full name of the PK generated with the local object mask.
<flexfield code>	Flexfield value for the primary key.

Examples

```
The primary key of my table is called: <%=odiRef.getPK("KEY_NAME")%>
```

A.2.45 getPKColList() Method

Use to return information about the columns of a primary key.

Usage

```
public java.lang.String getPKColList( java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Description

Returns a list of columns and expressions for the primary key being checked.

The pPattern parameter is interpreted and then repeated for each element of the list. It is separated from its predecessor by the pSeparator parameter. The generated string starts with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains an element for each column of the current primary key. It is accessible from a Check Knowledge Module if the current task is tagged as an "primary key".

In an action, this method returns the list of the columns of the primary key handled by the DDL command, ordered by their position in the key.

Parameters

Parameter	Type	Description
pStart	String	This sequence marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of attributes that can be used in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern sequence is replaced with its value. The attributes must be between brackets. ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This sequence marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as their associated description.

Parameter Value	Description
I_COL	Column internal identifier
COL_NAME	Name of the key column
COL_HEADING	Header of the key column
COL_DESC	Column description
POS	Position of the column
LONGC	Length (Precision) of the column
SCALE	Scale of the column
FILE_POS	Beginning position of the column (fixed file)
BYTES	Number of physical bytes of the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column
COL_MANDATORY	Mandatory character of the column. Valid values are: ■ 0: null authorized ■ 1: not null
CHECK_FLOW	Flow control flag for of the column. Valid values are: ■ 0: do not check ■ 1: check
CHECK_STAT	Static control flag of the column. Valid values are: ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column

Parameter Value	Description
COL_DEC_SEP	Decimal symbol for the column
REC_CODE_LIST	List of the record codes retained for the column
COL_NULL_IF_ERR	Processing flag for the column. Valid values are: ■ 0: Reject ■ 1: Set active trace to null ■ 2: Set inactive trace to null
DEF_VALUE	Default value for the column
EXPRESSION	Not used
CX_COL_NAME	Not used
ALIAS_SEP	Grouping symbol used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
<flexfield code>	Flexfield value for the current column.

Examples

If the CUSTOMER table has an primary key PK_CUSTOMER (CUST_ID, CUST_ NAME) and you want to generate the following code:

```
create table T_PK_CUSTOMER (CUST_ID numeric(10) not null, CUST_NAME
varchar(50) not null)
```

You can use the following code:

```
create table T_<%=odiRef.getPK("KEY_NAME")%>
<%=odiRef.getPKColList("(", "[COL_NAME] [DEST_CRE_DT] not null", ", ", ")")%>
```

Explanation: the getPKColList function will be used to generate the `(CUST_ID numeric(10) not null, CUST_NAME varchar(50) not null)` part, which starts and stops with a parenthesis and repeats the pattern (column, a data type, and not null) separated by commas for each column of the primary key. Thus

- the first parameter **"("** of the function indicates that we want to start the string with the string "("
- the second parameter **"[COL_NAME] [DEST_CRE_DT] not null"** indicates that we want to repeat this pattern for each column of the primary key. The keywords [COL_NAME] and [DEST_CRE_DT] reference valid keywords of the Pattern Attributes List table
- the third parameter **", "** indicates that we want to separate interpreted occurrences of the pattern with the string ", "

- the forth parameter **")"** of the function indicates that we want to end the string with the string ")"

A.2.46 getPop() Method

Use to return information about an interface.

Usage

```
public java.lang.String getPop(java.lang.String pPropertyName)
```

Description

This method returns information about the current interface. The list of available information is described in the pPropertyName values table.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the different possible values for pPropertyName:

Parameter Value	Description
I_POP	Internal number of the interface.
FOLDER	Name of the folder of the interface
POP_NAME	Name of the interface
IND_WORK_TARG	Position flag of the staging area.
LSCHEMA_NAME	Name of the logical schema which is the staging area of the interface
DESCRIPTION	Description of the interface
WSTAGE	Flag indicating the nature of the target datastore: ■ E - target datastore is an existing table (not a temporary table). ■ N - target datastore is a temporary table in the data schema. ■ W - target datastore is a temporary table in the work schema.
TABLE_NAME	Target table name
KEY_NAME	Name of the update key
DISTINCT_ROWS	Flag for doubles suppression
OPT_CTX	Name of the optimization context of the interface
TARG_CTX	Name of the execution context of the interface
MAX_ERR	Maximum number of accepted errors
MAX_ERR_PRCT	Error indicator in percentage
IKM	Name of the Integration Knowledge Module used in this interface.
LKM	Name of the Loading Knowledge Module specified to load data from the staging area to the target if a single-technology IKM is selected for the staging area.
CKM	Name of the Check Knowledge Module used in this interface.
HAS_JRN	Returns 1 if there is a journalized table in source of the interface, 0 otherwise.

Parameter Value	Description
PARTITION_NAME	Name of the partition or sub-partition selected for the target datastore. If no partition is selected, returns an empty string.
PARTITION_TYPE	Type of the partition or sub-partition selected for the target datastore. If no partition is selected, returns an empty string. ■ P: Partition ■ S: Sub-partition
<flexfield code>	Flexfield value for the interface.

Examples

```
The current interface is: <%=odiRef.getPop("POP_NAME")%> and runs on the logical
schema: <%=odiRef.getInfo("L_SCHEMA_NAME")%>
```

A.2.47 getPrevStepLog() Method

Use to return information about the previous step executed in the package.

Usage

```
public java.lang.String getPrevStepLog(java.lang.String pPropertyName)
```

Description

Returns information about the most recently executed step in a package. The information requested is specified through the pPropertyName parameter. If there is no previous step (for example, if the getPrevStepLog step is executed from outside a package), the exception "No previous step" is raised.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property about the previous step. See the list of valid properties below.

The following table lists the different possible values for pPropertyName:

Parameter Value	Description
SESS_NO	The number of the session.
NNO	The number of the step within a package. The first step executed is 0.
STEP_NAME	The name of the step.

Parameter Value	Description
STEP_TYPE	A code indicating the type of step. The following values may be returned: ■ F: Interface ■ VD: Variable declaration ■ VS: Set/Increment variable ■ VE: Evaluate variable ■ V: Refresh variable ■ T: Procedure ■ OE: OS command ■ SE: ODI Tool ■ RM: Reverse-engineer model ■ CM: Check model ■ CS: Check sub-model ■ CD: Check datastore ■ JM: Journalize model ■ JD: Journalize datastore
CONTEXT_NAME	The name of the context in which the step was executed.
MAX_ERR	The maximum number or percentage of errors tolerated.
MAX_ERR_PRCT	Returns 1 if the maximum number of errors is expressed as a percentage, 0 otherwise.
RUN_COUNT	The number of times this step has been executed.
BEGIN	The date and time that the step began.
END	The date and time that the step terminated.
DURATION	Time the step took to execute in seconds.
STATUS	Returns the one-letter code indicating the status with which the previous step terminated. The state R (Running) is never returned. ■ D: Done (success) ■ E: Error ■ Q: Queued ■ W: Waiting ■ M: Warning
RC	Return code. 0 indicates no error.
MESSAGE	Error message returned by previous step, if any. Blank string if no error.
INSERT_COUNT	Number of rows inserted by the step.
DELETE_COUNT	Number of rows deleted by the step.
UPDATE_COUNT	Number of rows updated by the step.
ERROR_COUNT	Number of erroneous rows detected by the step, for quality control steps.

Examples

```
Previous step '<%=odiRef.getPrevStepLog("STEP_NAME")%>' executed
in '<%=odiRef.getPrevStepLog("DURATION")%>' seconds.
```

A.2.48 getQuotedString() Method

Use to return a quoted string.

Usage

```
public java.lang.String getQuotedString(java.lang.String pString)
```

Description

This method returns a string surrounded with quotes. It preserves quotes and escape characters such as \n, \t that may appear in the string.

This method is useful to protect a string passed as a value in Java, Groovy or Jython code.

Parameters

Parameter	Type	Description
Parameter	Type	Description
pString	String	String that to be protected with quotes.

Examples

In the following Java code, the getQuotedString method is used to generate a valid string value.

```
String condSqlOK = <%=odiRef.getQuotedString(odiRef.getCK("MESS"))%>;
String condSqlKO = <%=odiRef.getCK("MESS")%>;
```

If the message for the condition is "*Error:\n Zero is not a valid value*", the generated code is as shown below. Without the getQuotedString, the code is incorrect, as the \n is not preserved and becomes a carriage return.

```
String condSqlOK = "Error:\n Zero is not a valid value";
String condSqlKO = "Error:
Zero is not a valid value";
```

A.2.49 getSchemaName() Method

Use to return a schema name from the topology.

Usage

```
public java.lang.String getSchemaName(
java.lang.String pLogicalSchemaName,
java.lang.String pLocation)

public java.lang.String getSchemaName(
java.lang.String pLogicalSchemaName,
java.lang.String pContextCode,
java.lang.String pLocation)

public java.lang.String getSchemaName( java.lang.String pLocation)


public java.lang.String getSchemaName()
```

Description

Retrieves the physical name of a data schema or work schema from its logical schema.

If the first syntax is used, the returned schema corresponds to the current context.

If the second syntax is used, the returned schema corresponds to context specified in the pContextCode parameter.

The third syntax returns the name of the data schema (D) or work schema (W) for the current logical schema in the current context.

The fourth syntax returns the name of the data schema (D) for the current logical schema in the current context.

Parameters

Parameter	Type	Description
pLogicalSchemaName	String	Name of the logical schema of the schema
pContextCode	String	Forced context of the schema
pLocation	String	The valid values are: ■ D: Returns the data schema of the physical schema that corresponds to the tuple (context, logical schema) ■ W: Returns the work schema of the physical schema that corresponds to the tuple (context, logical schema)

Examples

If you have defined the physical schema: Pluton.db_odi.dbo

Data catalog:	db_odi
Data schema:	dbo
Work catalog:	tempdb
Work schema:	temp_owner

and you have associated this physical schema to the logical schema: MSSQL_ODI in the context CTX_DEV

The Call To	Returns
`<%=odiRef.getSchemaName("MSSQL_ODI", "CTX_DEV", "W")%>`	`temp_owner`
`<%=odiRef.getSchemaName("MSSQL_ODI", "CTX_DEV", "D")%>`	`dbo`

A.2.50 getSchemaNameDefaultPSchema() Method

Use to return a catalog name for the default physical schema from the topology.

Usage

```
public java.lang.String getSchemaNameDefaultPSchema(
java.lang.String pLogicalSchemaName,
java.lang.String pLocation)

public java.lang.String getSchemaNameDefaultPSchema(
java.lang.String pLogicalSchemaName,
java.lang.String pContextCode,
```

```
java.lang.String pLocation)

public java.lang.String getSchemaNameDefaultPSchema(
java.lang.String pLocation)

public java.lang.String getSchemaNameDefaultPSchema(
```

Description

Allows you to retrieve the name of the **default** physical data schema or work schema for the data server to which is associated the physical schema corresponding to the tuple (logical schema, context). If no context is specified, the current context is used. If no logical schema name is specified, then the current logical schema is used. If no pLocation is specified, then the data schema is returned.

Parameters

Parameter	Type	Description
pLogicalSchemaName	String	Name of the logical schema
pContextCode	String	Code of the enforced context of the schema
pLocation	String	The valid values are: ■ D: Returns the data schema of the physical schema corresponding to the tuple (context, logical schema) ■ W: Returns the work schema of the default physical schema associate to the data server to which the physical schema corresponding to the tuple (context, logical schema) is also attached.

Examples

If you have defined the physical schemas: `Pluton.db_odi.dbo`

Data catalog:	db_odi
Data schema:	dbo
Work catalog:	tempdb
Work schema:	temp_odi
Default Schema	Yes

that you have associated with this physical schema: `MSSQL_ODI` in the context `CTX_DEV`, and `Pluton.db_doc.doc`

Data catalog:	db_doc
Data schema:	doc
Work catalog:	tempdb
Work schema:	temp_doc
Default Schema	No

that you have associated with this physical schema: MSSQL_DOC in the context CTX_DEV

The Call To	Returns
`<%=odiRef.getSchemaNameDefaultPSchema("MSSQL_DOC", "CTX_DEV", "W")%>`	`temp_odi`
`<%=odiRef.getSchemaNameDefaultPSchema("MSSQL_DOC", "CTX_DEV", "D")%>`	`dbo`

A.2.51 getSession() Method

Use to return information about the current session.

Usage

```
public java.lang.String getSession(java.lang.String pPropertyName)
```

Description

This method returns information about the current session. The list of available properties is described in the pPropertyName values table.

Parameters

Parameters	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the different possible values for pPropertyName:

Parameter Value	Description
SESS_NO	Internal number of the session
SESS_NAME	Name of the session
SCEN_NAME	Name of the scenario
SCEN_VERSION	Current scenario version
CONTEXT_NAME	Name of the execution context
CONTEXT_CODE	Code of the execution context
AGENT_NAME	Name of the physical agent in charge of the execution
SESS_BEG	Date and time of the beginning of the session
USER_NAME	ODI User running the session.

Examples

```
The current session is: <%=odiRef.getSession("SESS_NAME")%>
```

A.2.52 getSessionVarList() Method

Reserved for future use.

Usage

```
public java.lang.String getSessionVarList( java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd,
java.lang.String pSelector)
```

Description

Reserved for future use.

Parameters

Reserved for future use.

Examples

Reserved for future use.

A.2.53 getSrcColList() Method

Use to return properties for each column from a filtered list of source columns involved in a loading or integration phase. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getSrcColList(
java.lang.Int pDSIndex,
java.lang.String pStart,
java.lang.String pUnMappedPattern,
java.lang.String pMappedPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Description

This method available in LKMs and IKMs, returns properties for a list of columns in a given dataset. This list includes all the columns of the sources processed by the LKM (from the source) or the IKM (from the staging area). The list is sorted by the column position in the source tables.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

The properties displayed depend on whether the column is mapped or not. If the column is mapped, the properties returned are defined in the `pMappedPattern` pattern. If the column is not mapped, the properties returned are defined in the `pUnMappedPattern` pattern.

The attributes usable in a pattern are detailed in "Pattern Attributes List". Each occurrence of the attributes in the pattern string is replaced by its value. Attributes must be between brackets ([and]). Example: "My string [COL_NAME] is a column".

The `pMappedPattern` or `pUnMappedPattern` parameter is interpreted and then repeated for each element of the list. Patterns are separated with `pSeparator`. The generated string begins with `pStart` and ends with `pEnd`.

If there is a journalized datastore in the source of the interface, the three journalizing pseudo columns `JRN_FLG`, `JRN_DATE` and `JRN_SUBSCRIBER` are added as columns of the journalized source datastore.

Parameters

Parameter	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.
pStart	String	This sequence marks the beginning of the string to generate.
pUnMappedPattern	String	The pattern is repeated for each occurrence in the list if the column is not mapped.
pMappedPattern	String	The pattern is repeated for each occurrence in the list, if the column is mapped.
pSeparator	String	This parameter separates patterns.
pEnd	String	This sequence marks the end of the string to generate.

Pattern Attributes List

The following table lists different parameters values as well as their associated description.

Parameter Value	Description
I_COL	Internal identifier of the column
COL_NAME	Name of the column
ALIAS_NAME	Name of the column. Unlike COL_NAME, this attribute returns the column name without the optional technology delimiters. These delimiters appear when the column name contains for instance spaces.
COL_HEADING	Header of the column
COL_DESC	Description of the column
POS	Position of the column
LONGC	Column length (Precision)
SCALE	Scale of the column
FILE_POS	Beginning (index) of the column
BYTES	Number of physical bytes in the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column
COL_MANDATORY	Mandatory character of the column. Valid values are: (0: null authorized, 1: not null) ■ 0: null authorized ■ 1: not null
CHECK_FLOW	Flow control flag of the column. Valid values are: (0: do not check, 1: check) ■ 0: do not check ■ 1: check
CHECK_STAT	Static control flag of the column. Valid values are: (0: do not check, 1: check) ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column
COL_DEC_SEP	Decimal symbol of the column

Parameter Value	Description
REC_CODE_LIST	List of the record codes retained in the column
COL_NULL_IF_ERR	Processing flag of the column. Valid values are: ■ 0: Reject ■ 1: Set to null active trace ■ 2: Set to null inactive trace
DEF_VALUE	Default value of the column
EXPRESSION	Text of the expression (as typed in the mapping field) executed on the source (LKM) or the staging area (IKM). If the column is not mapped, this parameter returns an empty string.
CX_COL_NAME	Not supported.
ALIAS_SEP	Separator used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target (IKM) or staging area (LKM) technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
MANDATORY_CLAUSE	Returns `NOT NULL` if the column is mandatory. Otherwise, returns the null keyword for the technology.
DEFAULT_CLAUSE	Returns `DEFAULT <default value>` if any default value exists. Otherwise, returns and empty string.
<flexfield code>	Flexfield value for the current column.
POP_ALIAS	Alias of the datastore used in the interface.

Examples

To create a table similar to a source file:

```
create table <%=odiRef.getTable("L","COLL_NAME", "D")%>_F
(
<%=odiRef.getSrcColList("","[COL_NAME] [DEST_CRE_DT]","[COL_NAME]
[DEST_CRE_DT]",",\n","")%>
)
```

A.2.54 getSrcTablesList() Method

Use to return properties for each source table of an interface. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getSrcTablesList(
java.lang.Int pDSIndex,
java.lang.String pStart,
```

```
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getSrcTablesList(
java.lang.Int pDSIndex,
java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Returns a list of source tables of a given dataset in an interface. This method can be used to build a FROM clause in a SELECT order. However, it is advised to use the getFrom() method instead.

In IKMs only, the pDSIndex parameter identifies which of the datasets is taken into account by this command.

> **Note:** The pDSIndex parameter can be omitted when this method is used in an LKM. It can be also omitted for IKMs. In this case, the dataset taken into account is the first one.

The pPattern pattern is interpreted and then repeated for each element of the list and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameters	Type	Description
pDSIndex	Int	Index identifying which of the datasets is taken into account by this command.
pStart	String	This parameter marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of possible attributes in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern string is substituted with its value. The attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This parameter marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as the associated description.

Attribute	Description
I_TABLE	Internal identifier of the current source table if available.
MODEL_NAME	Name of the model of the current source table, if available.
SUB_MODEL_NAME	Name of the sub-model of the current source table, if available
TECHNO_NAME	Name of the technology of the source datastore
LSCHEMA_NAME	Logical schema of the source table
TABLE_NAME	Logical name of the source datastore
RES_NAME	Physical access name of the resource (file name or JMS queue, physical name of the table, etc.). If there is a journalized datastore in source of the interface, the source table is the clause is replaced by the data view linked to the journalized source datastore.
CATALOG	Catalog of the source datastore (resolved at runtime)
WORK_CATALOG	Work catalog of the source datastore
SCHEMA	Schema of the source datastore (resolved at runtime)
WORK_SCHEMA	Work schema of the source datastore
TABLE_ALIAS	Alias of the datastore as it appears in the tables list, if available
POP_TAB_ALIAS	Alias of the datastore as it appears in the current interface, if available.
TABLE_TYPE	Type of the datastore source, if available.
DESCRIPTION	Description of the source datastore, if available.
R_COUNT	Number of records of the source datastore, if available.
FILE_FORMAT	File format, if available.
FILE_SEP_FIELD	Field separator (file)
XFILE_SEP_FIELD	Hexadecimal field separator (file)
SFILE_SEP_FIELD	Field separator string (file)
FILE_ENC_FIELD	Field beginning and ending character (file)
FILE_SEP_ROW	Record separator (file)
XFILE_SEP_ROW	Hexadecimal record separator (file)
SFILE_SEP_ROW	Record separator string (file)
FILE_FIRST_ROW	Number of header lines to ignore, if available.
FILE_DEC_SEP	Default decimal separator for the datastore, if available.
METADATA	Description in ODI format of the metadata of the current resource, if available.
OLAP_TYPE	OLAP type specified in the datastore definition
IND_JRN	Flag indicating that the datastore is including in CDC.
JRN_ORDER	Order of the datastore in the CDC set for consistent journalizing.
PARTITION_NAME	Name of the partition or sub-partition selected for the source datastore. If no partition is selected, returns an empty string.
PARTITION_TYPE	Type of the partition or sub-partition selected for the source datastore. If no partition is selected, returns an empty string. ■ P: Partition ■ S: Sub-partition
<flexfield code>	Flexfield value for the current table.

Examples

```
insert into <%=odiRef.getTable("L", "COLL_NAME", "W")%>
select <%=odiRef.getColList("", "[EXPRESSION]", ", ", "", "INS=1")%>
from <%=odiRef.getSrcTablesList("", "[CATALOG].[SCHEMA].[TABLE_NAME] AS [POP_TAB_
ALIAS]", ", ", "")%>
where (1=1)
<%=odiRef.getJoinList("and ","([EXPRESSION])"," and ","")%>
<%=odiRef.getFilterList("and ","([EXPRESSION])"," and ","")%>
<%=odiRef.getGrpBy()%>
<%=odiRef.getHaving()%>
```

Explanation: the getSrcTablesList function will be used to generate the FROM clause of the SELECT STATEMENT that repeats the pattern (CATALOG.SCHEMA.TABLE_NAME as POP_TAB_ALIAS) separated by commas for each table in source.

- The first parameter **""** of the function indicates that we want do not want to start the string with any specific character.
- The second parameter **"[CATALOG].[SCHEMA].[TABLE_NAME] as [POP_TAB_ALIAS]"** indicates that we want to repeat this pattern for each source table. The keywords [CATALOG], [SCHEMA], [TABLE_NAME] and [POP_TAB_ALIAS] reference valid keywords of the table Pattern Attributes List
- The third parameter**", "** indicates that we want to separate each interpreted occurrence of the pattern with the string ", "
- The fourth parameter **""** of the function indicates that we want to end the string with no specific character

A.2.55 getStep() Method

Use to return information about the current step.

Usage

```
public java.lang.String getStep(java.lang.String pPropertyName)
```

Description

This method returns information about the current step. The list of available information is described in the pPropertyName values table.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the possible values for pPropertyName:

Parameter Value	Description
SESS_NO	Number of the session to which the step belongs.
NNO	Number of the step in the session
NB_RUN	Number of execution attempts
STEP_NAME	Step name

Parameter Value	Description
STEP_TYPE	Step type
CONTEXT_NAME	Name of the execution context
VAR_INCR	Step variable increment
VAR_OP	Operator used to compare the variable
VAR_VALUE	Forced value of the variable
OK_EXIT_CODE	Exit code in case of success
OK_EXIT	End the package in case of success
OK_NEXT_STEP	Next step in case of success.
OK_NEXT_STEP_NAME	Name of the next step in case of success
KO_RETRY	Number of retry attempts in case of failure.
KO_RETRY_INTERV	Interval between each attempt in case of failure
KO_EXIT_CODE	Exit code in case of failure.
KO_EXIT	End the package in case of failure.
KO_NEXT_STEP	Next step in case of failure.
KO_NEXT_STEP_NAME	Name of the next step in case of failure

Examples

```
The current step is: <%=odiRef.getStep("STEP_NAME")%>
```

A.2.56 getSubscriberList() Method

Use to return properties for each of the subscribers of a journalized table. The properties are organized according to a string pattern.

Usage

```
public java.lang.String getSubscriberList( java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Alternative syntax:

```
public java.lang.String getSubscriberList(
java.lang.String pPattern,
java.lang.String pSeparator,
```

Description

Returns a list of subscribers for a journalized table. The pPattern parameter is interpreted and then repeated for each element of the list, and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

In the alternative syntax, any parameters not set are set to an empty string.

Parameters

Parameter	Type	Description
pStart	String	This sequence marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of the attributes usable in a pattern is detailed in the Pattern Attributes List below. Each occurrence of the attributes in the pattern string is replaced by its value. Attributes must be between brackets ([and]) Example «My name is [SUBSCRIBER]»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This sequence marks the end of the string to generate.

Pattern Attributes List

The following table lists different parameters values as well as their associated description.

Parameter Value	Description
SUBSCRIBER	Name of the Subscriber

Examples

```
Here is list of Subscribers: <%=odiRef.getSubscriberList("\nBegin List\n", "-
[SUBSCRIBER]", "\n", "\nEnd of List\n")%>
```

A.2.57 getSysDate() Method

Use to return the system date of the machine running the session in a given format.

Usage

```
public java.lang.String getSysDate()

public java.lang.String getSysDate(pDateFormat)
```

Description

This method returns the system date of the machine running the session.

Parameters

Parameter	Type	Description
pDateFormat	String	Date format used to return the system date. This pattern should follow the Java Date and Time pattern. For more information, see http://download.oracle.com/javase/1.4.2/docs/api/java/text/SimpleDateFormat.html

Examples

```
Current year is: <%=odiRef.getSysDate("y")%>
```

A.2.58 getTable() Method

Use to return the fully qualified named of a table. This table may be a source or target table, or one of the temporary or infrastructure table handled by Oracle Data Integrator.

Usage

```
public java.lang.String getTable(
java.lang.String pMode,
java.lang.String pProperty,
java.lang.String pLocation)

public java.lang.String getTable(
java.lang.String pProperty,
java.lang.String pLocation)

public java.lang.String getTable(
java.lang.String pProperty)
```

Description

Allows the retrieval of the fully qualified name of temporary and permanent tables handled by Oracle Data Integrator.

Parameters

Parameters	Type	Description
pMode	String	"L": Uses the local object mask to build the complete path of the object. This value is used when pMode is not specified. "R": Uses the object mask to build the complete path of the object "A" Automatic: Defines automatically the adequate mask to use.

Parameters	Type	Description
pProperty	String	Parameter that indicates the name of the table to be built. The list of possible values is: ■ ID: Datastore identifier. ■ TARG_NAME: Full name of the target datastore. In actions, this parameter returns the name of the current table handled by the DDL command. If partitioning is used on the target datastore of an interface, this property automatically includes the partitioning clause in the datastore name. ■ COLL_NAME: Full name of the loading datastore. ■ INT_NAME: Full name of the integration datastore. ■ ERR_NAME: Full name of the error datastore. ■ CHECK_NAME: Name of the error summary datastore. ■ CT_NAME: Full name of the checked datastore. ■ FK_PK_TABLE_NAME: Full name of the datastore referenced by a foreign key. ■ JRN_NAME: Full name of the journalized datastore. ■ JRN_VIEW: Full name of the view linked to the journalized datastore. ■ JRN_DATA_VIEW: Full name of the data view linked to the journalized datastore. ■ JRN_TRIGGER: Full name of the trigger linked to the journalized datastore. ■ JRN_ITRIGGER: Full name of the Insert trigger linked to the journalized datastore. ■ JRN _UTRIGGER: Full name of the Update trigger linked to the journalized datastore. ■ JRN_DTRIGGER: Full name of the Delete trigger linked to the journalized datastore. ■ SUBSCRIBER_TABLE: Full name of the datastore containing the subscribers list. ■ CDC_SET_TABLE: Full name of the table containing list of CDC sets. ■ CDC_TABLE_TABLE: Full name of the table containing the list of tables journalized through CDC sets. ■ CDC_SUBS_TABLE: Full name of the table containing the list of subscribers to CDC sets. ■ CDC_OBJECTS_TABLE: Full name of the table containing the journalizing parameters and objects. ■ <flexfield_code>: Flexfield value for the current target table.
pLocation	String	W: Returns the full name of the object in the physical catalog and the physical work schema that corresponds to the current tuple (context, logical schema) D: Returns the full name of the object in the physical catalog and the physical data schema that corresponds to the current tuple (context, logical schema) A: Lets Oracle Data Integrator determine the default location of the object. This value is used if pLocation is not specified.

Examples

If you have defined a physical schema called Pluton.db_odi.dbo as shown below:

Data catalog:	db_odi
Data schema:	dbo
Work catalog:	tempdb

Work schema:	temp_owner
Local Mask:	%CATALOG.%SCHEMA.%OBJECT
Remote mask:	%DSERVER:%CATALOG.%SCHEMA.%OBJECT
Loading prefix:	CZ_
Error prefix:	ERR_
Integration prefix:	I$_

You have associated this physical schema to the logical schema called MSSQL_ODI in the context CTX_DEV and your working with a table is named CUSTOMER.

A Call To	Returns
`<%=odiRef.getTable("L", "COLL_NAME", "W")%>`	`tempdb.temp_owner.CZ_0CUSTOMER`
`<%=odiRef.getTable("R", "COLL_NAME", "D")%>`	`MyServer:db_odi.dbo.CZ_0CUSTOMER`
`<%=odiRef.getTable("L", "INT_NAME", "W")%>`	`tempdb.temp_owner.I$_CUSTOMER`
`<%=odiRef.getTable("R", "ERR_NAME", "D")%>`	`MyServer:db_odi.dbo.ERR_CUSTOMER`

A.2.59 getTargetColList() Method

Use to return information about the active columns of the target table of an interface. Active columns are those having an active mapping. To return information about all columns of the target table, including active and non-active columns, use the getAllTargetColList() Method.

Usage

```
public java.lang.String getTargetColList( java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd,
java.lang.String pSelector)
```

Alternative syntaxes:

```
public java.lang.String getTargetColList( java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)

public java.lang.String getTargetColList( java.lang.String pPattern,
java.lang.String pSeparator)
```

Description

Provides a list of columns for the interface's target table.

The pPattern parameter is interpreted and then repeated for each element of the list (selected according to pSelector parameter) and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

In the alternative syntaxes, any parameters not set are set to an empty string.

Parameters

Parameters	Type	Description
pStart	String	This sequence marks the beginning of the string to generate.
pPattern	String	The pattern is repeated for each occurrence in the list. The list of the attributes usable in a pattern is detailed in the Pattern Attributes List below. Each occurrence of the attributes in the pattern string is replaced by its value. Attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column of the target»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This sequence marks the end of the string to generate.
pSelector	String	String that designates a Boolean expression that allows to filter the elements of the initial list with the following format: <SELECTOR> <Operator> <SELECTOR> etc. Parenthesis are authorized. Authorized operators: 1. No: NOT or! 2. Or: OR or || 3. And: AND or && Example: (INS AND UPD) OR TRG The description of valid selectors is provided below.

Pattern Attributes List

The following table lists different parameters values as well as their associated description.

Parameter Value	Description
I_COL	Internal identifier of the column
COL_NAME	Name of the column
COL_HEADING	Header of the column
COL_DESC	Description of the column
POS	Position of the column
LONGC	Column length (Precision)
SCALE	Scale of the column
FILE_POS	Beginning (index) of the column
BYTES	Number of physical bytes in the column
FILE_END_POS	End of the column (FILE_POS + BYTES)
IND_WRITE	Write right flag of the column
COL_MANDATORY	Mandatory character of the column. Valid values are: ■ 0: null authorized ■ 1: not null

Parameter Value	Description
CHECK_FLOW	Flow control flag of the column. Valid values are: (0: do not check, 1: check) ■ 0: do not check ■ 1: check
CHECK_STAT	Static control flag of the column. Valid values are: (0: do not check, 1: check) ■ 0: do not check ■ 1: check
COL_FORMAT	Logical format of the column
COL_DEC_SEP	Decimal symbol of the column
REC_CODE_LIST	List of the record codes retained in the column
COL_NULL_IF_ERR	Processing flag of the column. Valid values are: (0 = Reject, 1 = Set to null active trace, 2= set to null inactive trace) ■ 0: Reject ■ 1: Set to null active trace ■ 2: Set to null inactive trace
DEF_VALUE	Default value of the column
ALIAS_SEP	Separator used for the alias (from the technology)
SOURCE_DT	Code of the column's datatype.
SOURCE_CRE_DT	Create table syntax for the column's datatype.
SOURCE_WRI_DT	Create table syntax for the column's writable datatype.
DEST_DT	Code of the column's datatype converted to a datatype on the target technology.
DEST_CRE_DT	Create table syntax for the column's datatype converted to a datatype on the target technology.
DEST_WRI_DT	Create table syntax for the column's writable datatype converted to a datatype on the target technology.
SCD_COL_TYPE	Behavior defined for the Slowly Changing Dimensions for this column in the data model.
MANDATORY_CLAUSE	Returns `NOT NULL` is the column is mandatory. Otherwise, returns the null keyword for the technology.
DEFAULT_CLAUSE	Returns `DEFAULT <default value>` if any default value exists. Otherwise, returns and empty string.
JDBC_TYPE	Data Services - JDBC Type of the column returned by the driver.
<flexfield code>	Flexfield value for the current column.

Selectors Description

Parameter Value	Description
INS	■ LKM: Not applicable ■ IKM: Only for mapping expressions marked with insertion ■ CKM: Not applicable
UPD	■ LKM: Not applicable ■ IKM: Only for the mapping expressions marked with update ■ CKM: Non applicable

Parameter Value	Description
TRG	■ LKM: Not applicable ■ IKM: Only for the mapping expressions executed on the target ■ CKM: Mapping expressions executed on the target.
NULL	■ LKM: Not applicable ■ IKM: All mapping expressions loading not nullable columns ■ CKM: All target columns that do not accept null values
PK	■ LKM: Not applicable ■ IKM: All mapping expressions loading the primary key columns ■ CKM: All the target columns that are part of the primary key
UK	■ LKM: Not applicable. ■ IKM: All the mapping expressions loading the update key column chosen for the current interface. ■ CKM: Not applicable.
REW	■ LKM: Not applicable. ■ IKM: All the mapping expressions loading the columns with read only flag not selected. ■ CKM: All the target columns with read only flag not selected.
MAP	■ LKM: Not applicable ■ IKM: Not applicable ■ CKM: Flow control: All columns of the target table loaded with expressions in the current interface Static control: All columns of the target table
SCD_SK	LKM, CKM, IKM: All columns marked SCD Behavior: Surrogate Key in the data model definition.
SCD_NK	LKM, CKM, IKM: All columns marked SCD Behavior: Natural Key in the data model definition.
SCD_UPD	LKM, CKM, IKM: All columns marked SCD Behavior: Overwrite on Change in the data model definition.
SCD_INS	LKM, CKM, IKM: All columns marked SCD Behavior: Add Row on Change in the data model definition.
SCD_FLAG	LKM, CKM, IKM: All columns marked SCD Behavior: Current Record Flag in the data model definition.
SCD_START	LKM, CKM, IKM: All columns marked SCD Behavior: Starting Timestamp in the data model definition.
SCD_END	LKM, CKM, IKM: All columns marked SCD Behavior: Ending Timestamp in the data model definition.
WS_INS	SKM: The column is flagged as allowing INSERT using Data Services.
WS_UPD	SKM: The column is flagged as allowing UDATE using Data Services.
WS_SEL	SKM: The column is flagged as allowing SELECT using Data Services.

Examples

```
create table TARGET_COPY <%=odiRef.getTargetColList("(", "[COL_NAME] [DEST_DT]
null", ", ", ")", "")%>
```

A.2.60 getTableName() Method

Use to return the name of the loading or integration table.

Usage

```
public java.lang.String getTableName(
java.lang.String pProperty)
```

Description

This method returns the name of the temporary table used for loading or integration. This name is not qualified.

Parameters

Parameters	Type	Description
pProperty	String	Parameter that indicates the name of the table to retreive. The list of possible values is: ■ INT_SHORT_NAME: Name of the integration table. ■ COLL_SHORT_NAME: Name of the loading table.

Examples

```
<%= odiRef.getTableName("COLL_SHORT_NAME") %>
```

A.2.61 getTargetTable() Method

Use to return information about the target table of an interface.

Usage

```
public java.lang.String getTargetTable(java.lang.String pPropertyName)
```

Description

This method returns information about the current target table. The list of available data is described in the pPropertyName values table.

In an action, this method returns information on the table being processed by the DDL command.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the possible values for pPropertyName:

Parameter Value	Description
I_TABLE	Internal identifier of the datastore
MODEL_NAME	Name of the model of the current datastore.
SUB_MODEL_NAME	Name of the sub-model of the current datastore.

Parameter Value	Description
TECHNO_NAME	Name of the target technology.
LSCHEMA_NAME	Name of the target logical schema.
TABLE_NAME	Name of the target datastore.
RES_NAME	Physical name of the target resource.
CATALOG	Catalog name.
WORK_CATALOG	Name of the work catalog.
SCHEMA	Schema name
WORK_SCHEMA	Name of the work schema.
TABLE_ALIAS	Alias of the current datastore.
TABLE_TYPE	Type of the datastore.
DESCRIPTION	Description of the current interface.
TABLE_DESC	Description of the current interface's target datastore. For a DDL command, description of the current table.
R_COUNT	Number of lines of the current datastore.
FILE_FORMAT	Format of the current datastore (file)
FILE_SEP_FIELD	Field separator (file)
XFILE_SEP_FIELD	Hexadecimal field separator (file)
SFILE_SEP_FIELD	Field separator string (file)
FILE_ENC_FIELD	Field beginning and ending character (file)
FILE_SEP_ROW	Record separator (file)
XFILE_SEP_ROW	Hexadecimal record separator (file)
SFILE_SEP_ROW	Record separator string (file)
FILE_FIRST_ROW	Number of lines to ignore at the beginning of the file (file)
FILE_DEC_SEP	Decimal symbol (file)
METADATA_DESC	Description of the metadata of the datastore (file)
OLAP_TYPE	OLAP type specified in the datastore definition
IND_JRN	Flag indicating that the datastore is including in CDC.
JRN_ORDER	Order of the datastore in the CDC set for consistent journalizing.
WS_NAME	Data Services - Name of the Web service generated for this datastore's model.
WS_NAMESPACE	Data Services - XML namespace of the web Service.
WS_JAVA_PACKAGE	Data Services - Java package generated for the web Service.
WS_ENTITY_NAME	Data Services - Entity name used for this datastore in the web service.
WS_DATA_SOURCE	Data Services - Datasource specified for this datastore's web service.
PARTITION_NAME	Name of the partition or sub-partition selected for the target datastore. If no partition is selected, returns an empty string.
PARTITION_TYPE	Type of the partition or sub-partition selected for the target datastore. If no partition is selected, returns an empty string. ■ P: Partition ■ S: Sub-partition

Parameter Value	Description
<flexfield code>	Flexfield value for the current table.

Examples

```
The current table is: <%=odiRef.getTargetTable("RES_NAME")%>
```

A.2.62 getTemporaryIndex() Method

Use to return information about a temporary index defined for optimizing a join or a filter in an interface.

Usage

```
public java.lang.String getTemporaryIndex(java.lang.String pPropertyName)
```

Description

This method returns information relative to a temporary index being created or dropped by an interface.

It can be used in a Loading or Integration Knowledge Module task if the Create Temporary Index option is set to On Source or On Target for this task.

Parameters

Parameter	Type	Description
pPropertyName	String	String containing the name of the requested property.

The following table lists the different possible values for pPropertyName.

Parameter Value	Description
IDX_NAME	Name of the index. This name is computed and prefixed with the temporary index prefix defined for the physical schema.
FULL_IDX_NAME	Fully qualified name of the index. On the target tab, this name is qualified to create the index in the work schema of the staging area. On the source tab, this name is qualified to create the index in the source default work schema (LKM) or in the work schema of the staging area (IKM).
COLL_NAME	Fully qualified name of the loading table for an LKM. This property does not apply to IKMs.
CATALOG	Catalog containing the table to be indexed.
SCHEMA	Schema containing the table to be indexed.
WORK_CATALOG	Work catalog for the table to be indexed.
WORK_SCHEMA	Work schema for the table to be indexed.
DEF_CATALOG	Default catalog containing the table to be indexed.
DEF_SCHEMA	Default schema containing the table to be indexed.
DEF_WORK_CATALOG	Default work catalog for the table to be indexed.
DEF_WORK_SCHEMA	Default work schema for the table to be indexed.
DEF_WORK_SCHEMA	Default work schema for the table to be indexed.

Parameter Value	Description
LSCHEMA_NAME	Logical schema of the table to be indexed.
TABLE_NAME	Name of the table to be indexed.
FULL_TABLE_NAME	Fully qualified name of the table to be indexed.
INDEX_TYPE_CODE	Code representing the index type.
INDEX_TYPE_CLAUSE	Clause for creating an index of this type.
POP_TYPE_CLAUSE	Type of the clause for which the index is generated: ■ J: Join. ■ F: Filter.
EXPRESSION	Expression of the join or filer clause. Use for debug purposes.

Examples

```
Create <%=odiRef.getTemporaryIndex (" [INDEX_TYPE_CLAUSE] index [FULL_IDX_NAME] on
[FULL_TABLE_NAME] " )%>
<%=odiRef.getTemporaryIndexColList("(", "[COL_NAME]", ", ", ")")%>
```

A.2.63 getTemporaryIndexColList() Method

Use to return information about the columns of a temporary index for an interface.

Usage

```
public java.lang.String getTemporaryIndexColList(java.lang.String pStart,
java.lang.String pPattern,
java.lang.String pSeparator,
java.lang.String pEnd)
```

Description

Returns a list of columns of a temporary index.

The parameter pPattern in interpreted and repeated for each element of the list, and separated from its predecessor with the parameter pSeparator. The generated string begins with pStart and ends with pEnd. If no element is selected, pStart and pEnd are omitted and an empty string is returned.

This list contains one element for each column of the temporary index.

It can be used in a Loading or Integration Knowledge Module task if the Create Temporary Index option is set to On Source or On Target for this task.

Parameters

Parameter	Type	Description
Parameter	Type	Description
pStart	String	This parameter marks the beginning of the string to generate.

Parameter	Type	Description
pPattern	String	The pattern is repeated for each occurrence in the list. The list of possible attributes in a pattern is detailed in the Pattern Attributes List below. Each attribute occurrence in the pattern string is substituted with its value. The attributes must be between brackets ([and]) Example «My string [COL_NAME] is a column»
pSeparator	String	This parameter separates each pattern from its predecessor.
pEnd	String	This parameter marks the end of the string to generate.

Pattern Attributes List

The following table lists the different values of the parameters as well as the associated description.

Parameter Value	Description
CX_COL_NAME	Computed name of the column used as a container for the current expression on the staging area
COL_NAME	Name of the column participating to the index.
POS	Position of the first occurrence of this column in the join or filter clause this index optimizes.

Examples

```
Create <%=odiRef.getTemporaryIndex (" [INDEX_TYPE_CLAUSE] index [FULL_IDX_NAME] on
[FULL_TABLE_NAME] " )%>
<%=odiRef.getTemporaryIndexColList("(", "[COL_NAME]", ", ", ")")%>
```

A.2.64 getUser() Method

Use to return information about the user running the current session.

Usage

```
public java.lang.String getUser(java.lang.String pPropertyName)
```

Description

This method returns information about the user executing the current session. The list of available properties is described in the pPropertyName values table.

Parameters

Parameter	Type	Description
pPropertyName	String	String that contains the name of the requested property.

The following table lists the different possible values for pPropertyName:

Parameter Value	Description
Parameter Value	Description
I_USER	User identifier
USER_NAME	User name
IS_SUPERVISOR	Boolean flag indicating if the user is supervisor (1) or not (0).

Examples

```
This execution is performed by <%=odiRef.getUser("USER_NAME")%>
```

A.2.65 hasPK() Method

Use to return whether if the current datastore has a primary key.

Usage

```
public java.lang.Boolean hasPK()
```

Description

This method returns a boolean. The returned value is true if the datastore for which a web service is being generated has a primary key.

This method can only be used in SKMs.

Examples

```
<% if (odiRef.hasPK()) { %>

         There is a PK :

          <%=odiRef.getPK("KEY_NAME")%> : <%=odiRef.getPKColList("{",
           "\u0022[COL_NAME]\u0022", ", ", "}")%>

<% } else {%>

         There is NO PK.

<% } %>
```

A.2.66 isColAttrChanged() Method

Use to return whether a column attribute or comment is changed.

Usage

```
public java.lang.Boolean
isColAttrChanged(java.lang.String pPropertyName)
```

Description

This method is usable in a column action for altering a column attribute or comment. It returns a boolean indicating if the column attribute passed as a parameter has changed.

Parameters

Parameter	Type	Description
pPropertyName	String	Attribute code (see below).

The following table lists the different possible values for pPropertyName

Parameter Value	Description
DATATYPE	Column datatype, length or precision change,
LENGTH	Column length change (for example, VARCHAR(10) changes to VARCHAR(12)).
PRECISION	Column precision change (for example, DECIMAL(10,3) changes to DECIMAL(10,4)).
COMMENT	Column comment change.
NULL_TO_NOTNULL	Column nullable attribute change from NULL to NOT NULL.
NOTNULL_TO_NULL	Column nullable attribute change from NOT NULL to NULL.
NULL	Column nullable attribute change.
DEFAULT	Column default value change.

Examples

```
<% if (odiRef.IsColAttrChanged("DEFAULT") ) { %>
          /* Column default attribute has changed. */
<% } %>
```

A.2.67 nextAK() Method

Use to move to the next alternate key for a datastore.

Usage

```
public java.lang.Boolean nextAK()
```

Description

This method moves to the next alternate key (AK) of the datastore for which a Web service is being generated.

When first called, this method returns true and positions the current AK to the first AK of the datastore. If there is no AK for the datastore, it returns false.

Subsequent calls position the current AK to the next AKs of the datastore, and return true. If the is no next AK, the method returns false.

This method can be used only in SKMs.

Examples

In the example below, we iterate of all the AKs of the datastore. In each iteration of the while loop, the `getAK` and `getAKColList` methods return information on the various AKs of the datastore.

```
<% while (odiRef.nextAK()) { %>
         <%=odiRef.getAK("KEY_NAME")%>
```

```
    Columns <%=odiRef.getAKColList("{", "\u0022[COL_NAME]\u0022", ",
 ", "}")%>
         Message : <%=odiRef.getAK("MESS")%>
<% } %>
```

A.2.68 nextCond() Method

Use to move to the next condition for a datastore.

Usage

```
public java.lang.Boolean nextCond()
```

Description

This method moves to the next condition (check constraint) of the datastore for which a Web service is being generated.

When first called, this method returns true and positions the current condition to the first condition of the datastore. If there is no condition for the datastore, it returns false.

Subsequent calls position the current condition to the next conditions of the datastore, and return true. If the is no next condition, the method returns false.

This method can be used only in SKMs.

Examples

In the example below, we iterate of all the conditions of the datastore. In each iteration of the while loop, the getCK method return information on the various conditions of the datastore.

```
<% while (odiRef.nextCond()) { %>
        <%=odiRef.getCK("COND_NAME")%>
                 SQL :<%=odiRef.getCK("COND_SQL")%>
                 MESS :<%=odiRef.getCK("MESS")%>
<% } %>
```

A.2.69 nextFK() Method

Use to move to the next foreign key for a datastore.

Usage

```
public java.lang.Boolean nextFK()
```

Description

This method moves to the next foreign key (FK) of the datastore for which a Web service is being generated.

When first called, this method returns true and positions the current FK to the first FK of the datastore. If there is no FK for the datastore, it returns false.

Subsequent calls position the current FK to the next FKs of the datastore, and return true. If the is no next FK, the method returns false.

This method can be used only in SKMs.

Examples

In the example below, we iterate of all the FKs of the datastore. In each iteration of the while loop, the `getFK` and `getFKColList` methods return information on the various FKs of the datastore.

```
<% while (odiRef.nextFK()) { %>
        FK : <%=odiRef.getFK("FK_NAME")%>
                Referenced Table : <%=odiRef.getFK("PK_TABLE_NAME")%>
                Columns <%=odiRef.getFKColList("{", "\u0022[COL_NAME]\u0022", ",
 ", "}")%>
                Message : <%=odiRef.getFK("MESS")%>
<% } %>
```

A.2.70 setNbInsert(), setNbUpdate(), setNbDelete(), setNbErrors() and setNbRows() Methods

Use to set the number of inserted, updated, deleted or erroneous rows for the current task.

Usage

```
public java.lang.Void setNbInsert(public java.lang.Long)

public java.lang.Void setNbUpdate(public java.lang.Long)

public java.lang.Void setNbDelete(public java.lang.Long)

public java.lang.Void setNbErrors(public java.lang.Long)

public java.lang.Void setNbRows(public java.lang.Long)
```

Description

These methods set for the current task report the values for:

- the number of rows inserted (setNbInsert)
- the number of rows updated (setNbUpdate)
- the number of rows deleted (setNbDelete)
- the number of rows in error (setNbErrors)
- total number of rows handled during this task (setNbRows)

These numbers can be set independently from the real number of lines processed.

> **Note:** This method can be used only within scripting engine commands, such as in Jython code, and should not be enclosed in <%%> tags.

Examples

In the Jython example below, we set the number of inserted rows to the constant value of 50, and the number of erroneous rows to a value coming from an ODI variable called `#DEMO.NbErrors`.

```
InsertNumber=50

odiRef.setNbInsert(InsertNumber)
```

```
ErrorNumber=#DEMO.NbErrors

odiRef.setNbErrors(ErrorNumber)
```

A.2.71 setTableName() Method

Use to set the name of the loading or integration table.

Usage

```
public java.lang.Void setTableName(
java.lang.String pProperty,
java.lang.String pTableName)
```

Description

This method sets the name of temporary table used for loading or integration. this name can be any value.

When using the method, the loading or integration table name is no longer generated by ODI and does not follow the standard naming convention (for example, a loading table will not be prefixed with a C$ prefix). Yet, other methods using this table name will return the newly set value.

The fist parameter pProperty indicates the temporary table name to set. The second parameter can be any valid table name.

Parameters

Parameters	Type	Description
pProperty	String	Parameter that indicates the table name to set. The list of possible values is: ■ INT_SHORT_NAME: Name of the integration table. ■ COLL_SHORT_NAME: Name of the loading table.
pTableName	String	New name for the temporary table.

Examples

```
<% odiRef.setTableName("COLL_SHORT_NAME", "C" + getInfo("I_SRC_SET")) %>

<% odiRef.setTableName("COLL_SHORT_NAME", odiRef.getOption("Flow # ") +
odiRef.getTable("ID")) %>
```

A.2.72 setTaskName() Method

Use to set the name of a session task in a Knowledge Module, Procedure, or action.

Usage

```
public java.lang.String setTaskName(
java.lang.String taskName)
```

Description

This method sets the name of a task to the `taskName` value. This value is set at run-time. This method is available in all Knowledge Modules, procedures, and actions (Global Methods).

Parameters

Parameters	Type	Description
`taskName`	String	Parameter that indicates the task name to set. If this vlaue is empty, the task remains the one defined in the Knowledge Module or Procedure task.

Examples

```
<$=odiRef.setTaskName("Create Error Table " + "<%=odiRef.getTable("L","ERR_
NAME","W")%>") $>
<$=odiRef.setTaskName("Insert Error for " + "<%=odiRef.getFK("FK_NAME")%>") $>
<$=odiRef.setTaskName("Loading " + "<%=odiRef.getTable("L", "COLL_NAME", "W")%>" +
" from " + "<%=odiRef.getSrcTablesList("", "RES_NAME", ",", ".")%>" ) $>
```

B

SNP_REV Tables Reference

This appendix provides a description of the Oracle Data Integrator SNP_REV tables. These tables are stored in a design-time repository and are used as staging tables for model metadata.

Customized Reverse-engineering processes load these tables before integrating their content into the repository tables describing the models.

See Chapter 3, "Reverse-Engineering Strategies" for more information.

B.1 SNP_REV_SUB_MODEL

SNP_REV_SUB_MODEL describes the sub-models hierarchy to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
SMOD_CODE	varchar(35)	Yes	Sub-model code
SMOD_NAME	varchar(400)	No	Sub-model name
SMOD_PARENT_CODE	varchar(35)	No	Parent sub-model code
IND_INTEGRATION	varchar(1)	No	Deprecated.
TABLE_NAME_PATTERN	varchar(35)	No	Automatic assignment mask used to distribute datastores in this sub-model
REV_APPY_PATTERN	varchar(1)	No	Datastores distribution rule: ■ 0: No distribution ■ 1: Automatic distribution of all datastores not already in a sub-model ■ 2: Automatic distribution of all datastores
REV_PATTERN_ORDER	varchar(10)	No	Order into which the pattern is applied.

B.2 SNP_REV_TABLE

SNP_REV_TABLE describes the datastores (tables, views, etc.) to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
TABLE_NAME	varchar(128)	Yes	Datastore name
RES_NAME	varchar(400)	No	Resource Name: Physical table or file name.

Column	Type	Mandatory	Description
TABLE_ALIAS	varchar(128)	No	Default datastore alias
TABLE_TYPE	varchar(2)	No	Datastore type: ■ T: Table or File ■ V: View ■ Q: Queue ■ AT: Table Alias ■ SY: Synonym ■ ST: System Table
TABLE_DESC	varchar(250)	No	Datastore description
IND_SHOW	varchar(1)	No	Datastore visibility: ■ 0: Hidden ■ 1: Displayed
R_COUNT	numeric(10)	No	Estimated row count
FILE_FORMAT	varchar(1)	No	Record format (applies only to files and JMS messages): ■ D: Delimited file ■ F: Fixed length file
FILE_SEP_FIELD	varchar(24)	No	Field separator (only applies to files and JMS messages)
FILE_ENC_FIELD	varchar(2)	No	Text delimiter (only applies to files and JMS messages)
FILE_SEP_ROW	varchar(24)	No	Record separator (only applies to files and JMS messages)
FILE_FIRST_ROW	numeric(10)	No	Number of header records to skip (only applies to files and JMS messages)
FILE_DEC_SEP	varchar(1)	No	Default decimal separator for numeric fields of the file (only applies to files and JMS messages)
SMOD_CODE	varchar(35)	No	Code of the sub-model containing this datastore. If null, the datastore is in the main model.
OLAP_TYPE	varchar(2)	No	OLAP Type: ■ DH: Slowly Changing Dimension ■ DI: Dimension ■ FA: Fact Table
WS_NAME	varchar(400)	No	Data service name.
WS_ENTITY_NAME	varchar(400)	No	Data service entity name.
SUB_PARTITION_METH	varchar(1)	No	Partitioning method: ■ H: Hash ■ R: Range ■ L: List
PARTITION_METH	varchar(1)	No	Sub-partitioning method: ■ H: Hash ■ R: Range ■ L: List

B.3 SNP_REV_COL

SNP_REV_COL lists the datastore columns to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
TABLE_NAME	varchar(128)	Yes	Datastore name
COL_NAME	varchar(128)	Yes	Column name
COL_HEADING	varchar(128)	No	Short description of the column
COL_DESC	varchar(250)	No	Long description of the column
DT_DRIVER	varchar(35)	No	Data type of the column. This data type should match the data type code as defined in Oracle Data Integrator Topology for this technology
POS	numeric(10)	No	Position of the column (not used for fixed length columns of files)
LONGC	numeric(10)	No	Logical length of the column (precision for numeric)
SCALEC	numeric(10)	No	Logical scale of the column
FILE_POS	numeric(10)	No	Starting position of the column (used only for fixed length files)
BYTES	numeric(10)	No	Number of physical bytes to read from file (not used for table columns)
IND_WRITE	varchar(1)	No	1/0 to indicate whether the column is writable.
COL_MANDATORY	varchar(1)	No	1/0 to indicate whether the column is mandatory.
CHECK_FLOW	varchar(1)	No	1/0 to indicate whether to include the mandatory constraint check by default in the static control.
CHECK_STAT	varchar(1)	No	1/0 to indicate whether to include the mandatory constraint check by default in the static control.
COL_FORMAT	varchar(35)	No	Column format. Typically this field applies only to files and JMS messages to define the date format.
COL_DEC_SEP	varchar(1)	No	Decimal separator for the column (applies only to files and JMS messages)
REC_CODE_LIST	varchar(250)	No	Record code to filter multiple record files (applies only to files and JMS messages)
COL_NULL_IF_ERR	varchar(1)	No	Indicate behavior in case of error with this column: ■ 0: Reject Error ■ 1: Null if error (inactive trace) ■ 2: Null if error (active trace)
DEF_VALUE	varchar(100)	No	Default value for this column.
SCD_COL_TYPE	varchar(2)	No	Slowly Changing Dimension type: ■ CR: Current Record Flag ■ ET: Ending Timestamp ■ IR: Add Row on Change ■ NK: Natural Key ■ OC: Overwrite on Change ■ SK: Surrogate Key ■ ST: Starting Timestamp

Column	Type	Mandatory	Description
IND_WS_SELECT	varchar(2)	No	0/1 to indicate whether this column is selectable using data services
IND_WS_UPDATE	varchar(2)	No	0/1 to indicate whether this column is updatable using data services
IND_WS_INSERT	varchar(2)	No	0/1 to indicate whether data can be inserted into this column using data services

B.4 SNP_REV_KEY

SNP_REV_KEY describes the datastore primary keys, alternate keys and indexes to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
TABLE_NAME	varchar(128)	Yes	Name of the datastore containing this constraint
KEY_NAME	varchar(128)	Yes	Key or index name
CONS_TYPE	varchar(2)	Yes	Key type: ■ PK: Primary key ■ AK: Alternate key ■ I: Index
IND_ACTIVE	varchar(1)	No	0/1 to indicate whether this constraint is active.
CHECK_FLOW	varchar(1)	No	1/0 to indicate whether to include this constraint check by default in the flow control.
CHECK_STAT	varchar(1)	No	1/0 to indicate whether to include this constraint check by default in the static control.

B.5 SNP_REV_KEY_COL

SNP_REV_KEY_COL lists the columns participating to the primary keys, alternate keys and indexes to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
TABLE_NAME	varchar(128)	Yes	Name of the datastore containing this constraint
KEY_NAME	varchar(128)	Yes	Key or index name
COL_NAME	varchar(128)	Yes	Name of the column in the key or index
POS	numeric(10)	No	Position of the column in the key

B.6 SNP_REV_JOIN

SNP_REV_JOIN describes the datastore references (foreign keys) to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
FK_NAME	varchar(128)	Yes	Reference (foreign key) name

Column	Type	Mandatory	Description
TABLE_NAME	varchar(128)	Yes	Name of the referencing table
FK_TYPE	varchar(1)	No	Reference type: ■ D: Database reference ■ U: User-defined reference ■ C: Complex user reference
PK_CATALOG	varchar(128)	No	Catalog of the referenced table (if different from the catalog of the referencing table)
PK_SCHEMA	varchar(128)	No	Schema of the referenced table (if different from the schema of the referencing table)
PK_TABLE_NAME	varchar(128)	No	Name of the referenced table
IND_ACTIVE	varchar(1)	No	0/1 to indicate whether this constraint is active.
CHECK_FLOW	varchar(1)	No	1/0 to indicate whether to include this constraint check by default in the flow control.
CHECK_STAT	varchar(1)	No	1/0 to indicate whether to include this constraint check by default in the static control.
DEFER	varchar(1)	No	Deferred constraint: ■ D: Deferrable ■ I: Immediate ■ N: Not Deferrable Not that this field is not used.
UPD_RULE	varchar(1)	No	On Update behavior: ■ C: Cascade ■ N: No Action ■ R: Restrict ■ S: Set Null
DEL_RULE	varchar(1)	No	On Delete behavior: ■ C: Cascade ■ N: No Action ■ R: Restrict ■ S: Set Null

B.7 SNP_REV_JOIN_COL

SNP_REV_JOIN_COL lists the matching columns participating to the references (foreign keys) to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
FK_NAME	varchar(128)	Yes	Reference (foreign key) name
FK_COL_NAME	varchar(128)	Yes	Name of the column in the referencing table
FK_TABLE_NAME	varchar(128)	No	Name of the referencing table
PK_COL_NAME	varchar(128)	Yes	Name of the column in the referenced table
PK_TABLE_NAME	varchar(128)	No	Name of the referenced table

Column	Type	Mandatory	Description
POS	numeric(10)	No	Position of the column in the reference

B.8 SNP_REV_COND

SNP_REV_COND describes the datastore condition and filters to reverse-engineer.

Column	Type	Mandatory	Description
I_MOD	numeric(10)	Yes	Model ID
TABLE_NAME	varchar(128)	Yes	Name of the datastore containing this constraint
COND_NAME	varchar(128)	Yes	Condition or check constraint name
COND_TYPE	varchar(1)	Yes	Condition type: ■ C: Oracle Data Integrator condition ■ D: Database condition ■ F: Filter
COND_SQL	varchar(250)	No	SQL expression for applying this condition or filter
COND_MESS	varchar(250)	No	Error message for this condition
IND_ACTIVE	varchar(1)	No	0/1 to indicate whether this constraint is active.
CHECK_FLOW	varchar(1)	No	1/0 to indicate whether to include this constraint check by default in the flow control.
CHECK_STAT	varchar(1)	No	1/0 to indicate whether to include this constraint check by default in the static control.