# Oracle® Database

2 Day Developer's Guide

11*g* Release 2 (11.2)

**E10766-01**

July 2009

ORACLE®

Oracle Database 2 Day Developer's Guide, 11*g* Release 2 (11.2)

E10766-01



Primary Author: Sheila Moore

Contributors: Pat Huey, Sharon Kennedy, Simon Law, Roza Leyderman, Bryn Llewellen, Chuck Murray, Mark Townsend

# Contents

## 5 About DML Statements and Transactions



## 6 Creating and Managing Schema Objects

## 7 Developing Stored Subprograms and Packages

## 8 Using Triggers



## 9 Working in a Global Environment

## 10 Deploying an Oracle Database Application



## Index

## List of Examples

## List of Tables

# Preface

This document explains basic concepts behind application development with Oracle Database. It provides instructions for using the basic features of topics through Structured Query Language (SQL), and the Oracle server-based procedural extension to the SQL database language, Procedural Language/Structured Query Language (PL/SQL).

Preface topics:



## Audience

This document is intended for anyone who is interested in learning about Oracle Database application development, and is primarily an introduction to application development for developers who are new to Oracle Database.

This document assumes that you have a general understanding of relational database concepts and an understanding of the operating system environment that you will use to develop applications with Oracle Database.

## Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

### Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

### Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

### Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

## Related Documents

As you become comfortable with the concepts and tasks in this document, Oracle recommends that you consult other Oracle Database development documents, especially:

- *Oracle Database 2 Day + Application Express Developer's Guide*
- *Oracle Database 2 Day + Java Developer's Guide*
- *Oracle Database 2 Day + .NET Developer's Guide*
- *Oracle Database 2 Day + PHP Developer's Guide*

For more information, see:

- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Database Concepts*
- *Oracle Database SQL Language Reference*
- *Oracle Database PL/SQL Language Reference*

## Conventions

This document uses these text conventions:

| 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

This chapter contains the following topics:



## About Oracle Database Developers

An Oracle Database developer is responsible for creating or maintaining the database components of an application that uses the Oracle technology stack. Oracle Database developers either develop applications or convert existing applications to run in the Oracle Database environment.

> **See Also:** *Oracle Database Concepts* for more information about the duties of Oracle Database developers

## About This Document

This document is the entry into the Oracle Database documentation set for application developers. It does the following:

- Explains the basic concepts behind development with Oracle Database
- Shows how to use basic features of SQL and PL/SQL
- Provides references to detailed information about subjects that it introduces
- Shows, with tutorials and examples, how to develop and deploy a sample application

Chapter 1, "Introduction," describes the reader for whom this document is intended, outlines the organization of this document, introduces important Oracle Database concepts, and describes the sample schema used in the tutorials and examples in this document.

Chapter 2, "Connecting to Oracle Database," explains how to connect to Oracle Database.

Chapter 3, "Exploring Oracle Database with SQL Developer," shows how to view schema objects and the properties and data of Oracle Database tables.

Chapter 4, "Selecting Table Data," shows how to use queries to retrieve data from an Oracle Database table.

Chapter 5, "About DML Statements and Transactions," introduces data manipulation language (DML) statements and transactions. DML statements add, change, and delete Oracle Database table data. A transaction is a sequence of one or more SQL statements that Oracle Database treats as a unit: either all of the statements are performed, or none of them are.

Chapter 6, "Creating and Managing Schema Objects," introduces data definition language (DDL) statements, which create, change, and drop schema objects. The tutorials and examples show how to create the objects for the sample application.

Chapter 7, "Developing Stored Subprograms and Packages," introduces stored subprograms and packages, which can be used as building blocks for many different database applications. The tutorials and examples show how to create the package of subprograms for the sample application.

Chapter 8, "Using Triggers," introduces triggers, which are stored PL/SQL units that automatically execute ("fire") in response to specified events. The tutorials and examples show how to create the triggers for the sample application.

Chapter 9, "Working in a Global Environment," introduces globalization support—National Language Support (NLS) parameters and Unicode-related features of SQL and PL/SQL.

Chapter 10, "Deploying an Oracle Database Application," explains how to deploy a database application—that is, how to install it in one or more environments where other users can run it—using the sample application as an example.

# About Oracle Database

Oracle Database groups related information into logical structures called **schemas**. The logical structures are called **schema objects**. When you connect to the database by providing your user name and password, you specify the schema and indicate that you are its owner. In Oracle Database, the user name and the name of the schema to which the user connects are the same.

Topics:



## About Schema Objects

Every object in an Oracle Database belongs to only one schema, and has a unique name with that schema.

Some of the objects that schemas can contain are:

- **Tables**

  Tables are the basic units of data storage in Oracle Database. Tables hold all user-accessible data. Each table contains **rows** that represent individual data **records**. Rows are composed of **columns** that represent the **fields** of the records. For more information, see "Creating and Managing Tables" on page 6-2.

- **Indexes**

  Indexes are optional objects that can improve the performance of data retrieval from tables. Indexes are created on one or more columns of a table, and are automatically maintained in the database. For more information, see "Managing Indexes" on page 6-14.

- **Views**

  You can create a view that combines information from several different tables into a single presentation. A view can rely on information from both tables and other views. For more information, see "Creating and Managing Views" on page 6-17.

- **Sequences**

  When all records of a table must be distinct, you can use a sequence to generate a serial list of unique integers for numeric columns, each of which represents the ID of one record. For more information, see "Creating and Managing Sequences" on page 6-20.

- **Synonyms**

  Synonyms are aliases for schema objects. You can use synonyms for security and convenience; for example, to hide the ownership of an object or to simplify SQL statements. For more information, see "Creating and Managing Synonyms" on page 6-22.

- **Stored subprograms**

  Stored subprograms (also called **schema-level subprograms**) are procedures and functions that are stored in the database. They can be invoked from client applications that access the database. For more information, see "Developing Stored Subprograms and Packages" on page 7-1.

  **Triggers** are stored subprograms that are automatically run by the database when specified events occur in a particular table or view. Triggers can restrict access to specific data, perform logging, or audit data. For more information, see "Using Triggers" on page 8-1.

- **Packages**

  A package is a group of related subprograms, along with the explicit cursors and variables they use, stored in the database as a unit, for continued use. Like stored subprograms, packaged subprograms can be invoked from client applications that access the database. For more information, see "Developing Stored Subprograms and Packages" on page 7-1.

Typically, the objects that an application uses belong to the same schema.

> **See Also:** *Oracle Database Concepts* for a comprehensive introduction to schema objects

## About Oracle Database Access

You can access Oracle Database only through a client program, such as SQL*Plus or SQL Developer. The client program's interface to Oracle Database is Structured Query Language (SQL). Oracle provides an extension to SQL called Procedural Language/SQL (PL/SQL).

Topics:

### About SQL*Plus

**SQL*Plus** (pronounced *sequel plus*) is an interactive and batch query tool that is installed with every Oracle Database installation. It has a command-line user interface that acts as the client when connecting to the database.

SQL*Plus has its own commands and environment. In the SQL*Plus environment, you can enter and run SQL*Plus commands, SQL statements, PL/SQL statements, and operating system commands to perform tasks such as:

- Formatting, performing calculations on, storing, and printing query results
- Examining tables and object definitions
- Developing and running batch scripts
- Performing database administration

You can use SQL*Plus to generate reports interactively, to generate reports as batch processes, and to output the results to text file, to screen, or to HTML file for browsing on the Internet. You can generate reports dynamically using the HTML output facility.

You can use SQL*Plus in SQL Developer. For details, see *Oracle Database SQL Developer User's Guide*.

**See Also:**

- "Connecting to Oracle Database from SQL*Plus" on page 2-1
- *SQL*Plus User's Guide and Reference* for complete information about SQL*Plus

### About SQL Developer

**SQL Developer** (pronounced *sequel developer*) is a graphic version of SQL*Plus, written in Java, that is available in the default installation of Oracle Database and by free download.

The SQL Developer user interface includes a navigation frame, tools (with menus), and a SQL Worksheet. From the SQL Worksheet, you can enter and run SQL statements, PL/SQL statements, and SQL*Plus commands. You can do some tasks—for example, creating a table—either in the SQL Worksheet or with the navigation frame and tools.

To see the name and keyboard equivalent of any SQL Developer icon, position your cursor over the icon.

**See Also:**

- "Connecting to Oracle Database from SQL Developer" on page 2-2
- *Oracle Database SQL Developer User's Guide* for complete information about SQL Developer

### About Structured Query Language (SQL)

**Structured Query Language (SQL)** (pronounced *sequel*) is the set-based, high-level computer language with which all programs and users access data in Oracle Database.

SQL is a declarative, or nonprocedural, language; that is, it describes what to do, but not how. You specify the desired result set (for example, the names of current employees), but not how to get it.

**See Also:**

- *Oracle Database Concepts* for a complete overview of SQL
- *Oracle Database SQL Language Reference* for complete information about SQL

### About Procedural Language/SQL (PL/SQL)

**Procedural Language/SQL (PL/SQL)** (pronounced *P L sequel*) is a native Oracle Database extension to SQL. It bridges the gap between declarative and imperative program control by adding procedural elements, such as conditional control and loops.

In PL/SQL, you can declare constants and variables, procedures and functions, types and variables of those types, and triggers. You can handle exceptions (run-time errors). You can create PL/SQL units—procedures, functions, packages, types, and triggers—that are stored in the database for reuse by applications that use any of the Oracle Database programmatic interfaces.

The basic unit of a PL/SQL source program is the block, which groups related declarations and statements. A block has an optional declarative part, a required executable part, and an optional exception-handling part.

**See Also:**

- *Oracle Database Concepts* for a complete overview of PL/SQL
- *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL

### About Other Client Programs, Languages, and Development Tools

Some other database access clients, languages, and tools that you can use to develop applications are:

Topics:

- Oracle Application Express (APEX)
- Oracle Java Database Connectivity (JDBC)
- Hypertext Preprocessor (PHP)
- Oracle Call Interface (OCI)
- Oracle C++ Call Interface (OCCI)
- Open Database Connectivity (ODBC)
- Pro*C/C++ Precompiler
- Pro*COBOL Precompiler
- Microsoft .NET Framework
- Oracle Provider for OLE DB (OraOLEDB)
- Oracle Objects for OLE (OO4O)

**See Also:**

- *Oracle Database Concepts* for more information about tools for Oracle Database developers
- *Oracle Database Advanced Application Developer's Guide* for information about choosing a programming environment

**Oracle Application Express (APEX)** Oracle Application Express (APEX) is an application development and deployment tool that enables you to quickly create secure and scalable Web applications even if you have limited previous programming experience. The embedded Application Builder tool assembles an HTML interface or a complete application that uses schema objects, such as tables or stored procedures, into a collection of pages that are linked through tabs, buttons, or hypertext links.

**See Also:** *Oracle Database 2 Day + Application Express Developer's Guide* for more information about APEX

**Oracle Java Database Connectivity (JDBC)** Oracle Java Database Connectivity (JDBC) is an API that enables Java to send SQL statements to an object-relational database, such as Oracle Database. Oracle Database JDBC provides complete support for the JDBC 3.0 and JDBC RowSet (JSR-114) standards, advanced connection caching for both XA and non-XA connections, exposure of SQL and PL/SQL data types to Java, and fast SQL data access.

**See Also:** For more information about JDBC:

- *Oracle Database Concepts*
- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Database 2 Day + Java Developer's Guide*

**Hypertext Preprocessor (PHP)** The Hypertext Preprocessor (PHP) is a powerful interpreted server-side scripting language for quick Web application development. PHP is an open source language that is distributed under a BSD-style license. PHP is designed for embedding database access requests directly into HTML pages.

**See Also:** *Oracle Database 2 Day + PHP Developer's Guide* for more information about PHP

**Oracle Call Interface (OCI)** Oracle Call Interface (OCI) is the native C language API for accessing Oracle Database directly from C applications.

The OCI Software Development Kit is also installed as part of the Oracle Instant Client, which enables you to run applications without installing the standard Oracle client or having an `ORACLE_HOME`. Your applications work without change, using significantly less disk space.

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for more information about OCI
- *Oracle Call Interface Programmer's Guide* for complete information about OCI

**Oracle C++ Call Interface (OCCI)** Oracle C++ Call Interface (OCCI) is the native C++ language API for accessing Oracle Database directly from C++ applications. Very

similar to the OCI, OCCI supports both relational and object-oriented programming paradigms.

The OCCI Software Development Kit is also installed as part of the Oracle Instant Client, which enables you to run applications without installing the standard Oracle client or having an `ORACLE_HOME`. Your applications work without change, using significantly less disk space.

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for more information about OCCI
- *Oracle C++ Call Interface Programmer's Guide* for complete information about OCCI

**Open Database Connectivity (ODBC)** Open Database Connectivity (ODBC) is a set of database access APIs that connect to the database, prepare, and then run SQL statements on the database. An application that uses an ODBC driver can access non-uniform data sources, such as spreadsheets and comma-delimited files.

The Oracle ODBC driver conforms to ODBC 3.51 specifications. It supports all core APIs and a subset of Level 1 and Level 2 functions. Microsoft supplies the Driver manager component for the Windows platform.

Like OCI, OCCI, and JDBC, ODBC is part of the Oracle Instant Client installation.

**See Also:**

- *Oracle Database Concepts*
- *Oracle Services for Microsoft Transaction Server Developer's Guide* for information about using the Oracle ODBC driver with Windows
- *Oracle Database Administrator's Reference for Linux and UNIX-Based Operating Systems* for information about using Oracle ODBC driver on Linux

**Pro*C/C++ Precompiler** The Pro*C/C++ precompiler enables you to embed SQL statements in a C or C++ source file. The precompiler accepts the source program as input, translates the embedded SQL statements into standard Oracle run-time library calls, and generates a modified source program that you can compile, link, and run.

**See Also:**

- *Oracle Database Concepts* for more information about Oracle precompilers
- *Oracle Database Advanced Application Developer's Guide* for more information about the Pro*C/C++ precompiler
- *Pro*C/C++ Programmer's Guide* for complete information about the Pro*C/C++ precompiler

**Pro*COBOL Precompiler** The Pro*COBOL precompiler enables you to embed SQL statements in a COBOL source file. The precompiler accepts the source program as input, translates the embedded SQL statements into standard Oracle run-time library calls, and generates a modified source program that you can compile, link, and run.

**See Also:**

- *Oracle Database Concepts* for more information about Oracle precompilers
- *Oracle Database Advanced Application Developer's Guide* for more information about the Pro*COBOL precompiler
- *Pro*COBOL Programmer's Guide* for complete information about the Pro*COBOL precompiler

#### Microsoft .NET Framework

The Microsoft .NET Framework is a multilanguage environment for building, deploying, and running applications and XML Web services. Its main components are:

- Common Language Runtime (CLR)

  The Common Language Runtime (CLR) is a language-neutral development and run-time environment that provides services that help manage running applications.

- Framework Class Libraries (FCL)

  The Framework Class Libraries (FCL) provide a consistent, object-oriented library of prepackaged functionality.

Oracle Data Provider for .NET (ODP.NET)

Oracle Data Provider for .NET (ODP.NET) provides fast and efficient ADO.NET data access from .NET client applications to Oracle Database and access to other Oracle Database features.

ODP.NET allows developers to take advantage of advanced Oracle Database functionality, including Real Application Clusters, XML DB, and advanced security.

**Oracle Developer Tools for Visual Studio (ODT)**

Oracle Developer Tools for Visual Studio (ODT) is a set of application tools that integrate with the Visual Studio environment. These tools provide graphic user interface access to Oracle functionality, enable the user to perform a wide range of application development tasks, and improve development productivity and ease of use. Oracle Developer Tools supports the programming and implementation of .NET stored procedures using Visual Basic, C#, and other .NET languages.

**.NET Stored Procedures**

Oracle Database Extensions for .NET is a database option for Oracle Database on Windows. It makes it possible to build and run .NET stored procedures or functions with Oracle Database for Microsoft Windows using Visual Basic .NET or Visual C#.

After building .NET procedures and functions into a .NET assembly, you can deploy them in Oracle Database using the Oracle Deployment Wizard for .NET, a component of the Oracle Developer Tools for Visual Studio.

**Oracle Providers for ASP.NET**

Oracle Providers for ASP.NET offer ASP.NET developers an easy way to store state common to Web applications within Oracle Database. These providers are modeled on existing Microsoft ASP.NET providers, sharing similar schema and programming interfaces to provide .NET developers a familiar interface. Oracle supports the Membership, Profile, Role, and other providers.

**See Also:**

- *Oracle Database 2 Day + .NET Developer's Guide*
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows*
- *Oracle Database Extensions for .NET Developer's Guide for Microsoft Windows*
- *Oracle Database Advanced Application Developer's Guide*

**Oracle Provider for OLE DB (OraOLEDB)** Oracle Provider for OLE DB (OraOLEDB) is an open standard data access methodology that uses a set of Component Object Model (COM) interfaces for accessing and manipulating different types of data. These interfaces are available from various database providers.

**See Also:** *Oracle Provider for OLE DB Developer's Guide* for more information about OraOLEDB

**Oracle Objects for OLE (OO4O)** Oracle Objects for OLE (OO4O) provides easy access to data stored in Oracle Database with any programming or scripting language that supports the Microsoft COM Automation and ActiveX technology, including Visual Basic, Visual C++, Visual Basic For Applications (VBA), IIS Active Server Pages (VBScript and JavaScript).

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for more information about OO4O
- Oracle Objects for OLE Help, and Oracle Objects for OLE C++ Class Library Help, for detailed information about Oracle Objects for OLE
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for examples of how to use Oracle Objects for OLE
- *Oracle Streams Advanced Queuing User's Guide* for examples of how to use Oracle Objects for OLE

## About Sample Schema HR

The `HR` schema is a sample schema that can be installed as part of Oracle Database. This schema contains information about employees—their departments, locations, work histories, and related information. Like all schemas, the `HR` schema has tables, views, indexes, procedures, functions, and other attributes of a database schema.

The examples and tutorials in this document use the `HR` schema.

**See Also:**

- *Oracle Database Sample Schemas* for a complete description of the `HR` schema
- "Connecting to Oracle Database as User HR" on page 2-4 for instructions for connecting to Oracle Database as the user `HR`

# 2

# Connecting to Oracle Database

You can connect to Oracle Database only through a client program, such as SQL*Plus or SQL Developer.

This chapter contains the following sections:



## Connecting to Oracle Database from SQL*Plus

SQL*Plus is a client program with which you can access Oracle Database. This section shows how to start SQL*Plus and connect to Oracle Database.

> **Note:** For steps 3 and 4 of the following procedure, you need a user name and password.

**To connect to Oracle Database from SQL*Plus:**

1. If you are on a Windows system, display a Windows command prompt.
2. At the command prompt, type `sqlplus` and press the key **Enter**.

   SQL*Plus starts and prompts you for your user name.
3. Type your user name and press the key **Enter**.

   SQL*Plus prompts you for your password.
4. Type your password and press the key **Enter**.

   > **Note:** For security, your password is not visible on your screen.

   The system connects you to an Oracle Database instance.

   You are in the SQL*Plus environment. At the `SQL>` prompt, you can enter and run SQL*Plus commands, SQL statements, PL/SQL statements, and operating system commands.

   To exit SQL*Plus, type `exit` and press the key **Enter**.

**Note:** Exiting SQL*Plus ends the SQL*Plus session, but does not shut down the Oracle Database instance.

Example 2–1 starts SQL*Plus, connects to Oracle Database, runs a SQL `SELECT` statement, and exits SQL*Plus. User input is **bold**.

***Example 2–1 Connecting to Oracle Database from SQL*Plus***

```
> sqlplus
SQL*Plus: Release 11.2.0.0.1 - Beta on Mon Jun 9 15:31:26 2008

Copyright (c) 1982, 2008, Oracle.  All rights reserved.

Enter user-name: your_user_name
Enter password: your_password

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.0.0 - Development
With the Partitioning, Data Mining and Real Application Testing options

SQL> select count(*) from employees;

  COUNT(*)
----------
       107

SQL> exit

Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.0.0 - Development
With the Partitioning, Data Mining and Real Application Testing options
>
```

**See Also:**

- "Connecting to Oracle Database as User HR from SQL*Plus" on page 2-5
- "About SQL*Plus" on page 1-4 for a brief description of SQL*Plus
- *SQL*Plus User's Guide and Reference* for more information about starting SQL*Plus and connecting to Oracle Database

# Connecting to Oracle Database from SQL Developer

SQL Developer is a client program with which you can access Oracle Database. This section assumes that SQL Developer is installed on your system, and shows how to start it and connect to Oracle Database. If SQL Developer is not installed on your system, see *Oracle Database SQL Developer User's Guide* for installation instructions.

**Note:** For the following procedure:

- The first time you start SQL Developer on your system, you must provide the full path to `java.exe` in step 4.
- For step 7, you need a user name and password.
- For step 9, you need a host name and port.

**To connect to Oracle Database from SQL Developer:**

1. Access the menu from which you can select SQL Developer:
   - On a Windows system: From the Start menu, select **All Programs**.
   - On a Linux system:
     - On Gnome: Click **Application**.
     - On KDE: Click **K**.
2. Select **Oracle - *ORACLE_HOME***.
3. Select **Application Development**.
4. Select **SQL Developer**.

   If this is the first time you have started SQL Developer on your system, you are prompted to enter the full path to `java.exe` (for example, `C:\jdk1.5.0\bin\java.exe`). Either type the full path after the prompt or browse to it, and then press the key **Enter**.

   The Oracle SQL Developer window opens.
5. In the navigation frame of the window, click **Connections**.

   The Connections pane appears.
6. In the Connections pane, click the icon **New Connection**.

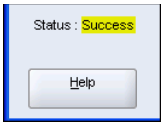
   The New/Select Database Connection window opens.
7. In the New/Select Database Connection window, type the appropriate values in the fields Connection Name, Username, and Password.

   For security, the password characters that you type appear as asterisks.

   Near the Password field is the check box Save Password. By default, it is deselected. Oracle recommends accepting the default.
8. In the New/Select Database Connection window, click the tab **Oracle**.

   The Oracle pane appears.
9. In the Oracle pane:
   - For Role, accept the default.
   - For Connection Type, accept the default (Basic).
   - In the fields Hostname and Port, type the appropriate values.
   - Select the option **SID**.
   - In the SID field, type the appropriate value.
10. In the New/Select Database Connection window, click the button **Test**.

    The connection is tested. If the connection succeeds, the Status indicator changes from blank to `Success`.

    
    
11. If the test succeeded, click the button **Connect**.

The New/Select Database Connection window closes. The Connections pane shows the connection whose name you entered in the Connection Name field in step 7.

You are in the SQL Developer environment.

To exit SQL Developer, select **Exit** from the File menu.

---

**Note:** Exiting SQL Developer ends the SQL Developer session, but does not shut down the Oracle Database instance. The next time you start SQL Developer, the connection you created using the preceding procedure still exists. SQL Developer prompts you for the password that you supplied in step 7 (unless you selected the check box Save Password).

---

**See Also:**

- "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6
- "About SQL Developer" on page 1-4 for a brief description of SQL Developer
- *Oracle Database SQL Developer User's Guide* for more information about using SQL Developer to create connections to Oracle Database

# Connecting to Oracle Database as User HR

This section shows how to unlock the `HR` account and connect to Oracle Database as the user `HR`, who owns the `HR` sample schema that the examples and tutorials in this document use.

To do the tutorials and examples in this document, and create the sample application, you must connect to Oracle Database as the user `HR` from SQL Developer. The `HR` sample schema is the **development environment** for the sample application.

Topics:



## Unlocking the HR Account

By default, when the `HR` schema is installed, the `HR` account is locked and its password is expired. You can connect to Oracle Database as the user `HR` only if the `HR` account is unlocked.

---

**Note:** For the following procedure, you need the name and password of a user who has the `ALTER USER` system privilege (for example, `SYSTEM`).

---

**To unlock the HR account and reset its password:**

1. Using SQL*Plus, connect to Oracle Database as a user with the `ALTER USER` system privilege.

   For instructions, see "Connecting to Oracle Database from SQL*Plus" on page 2-1.

2. At the `SQL>` prompt, unlock the `HR` account and reset its password:

   **Caution:** Choose a secure password. For guidelines for securing passwords, see *Oracle Database Security Guide*.

   ```
   ALTER USER HR ACCOUNT UNLOCK IDENTIFIED BY password;
   ```

   The system responds:

   ```
   User altered
   ```

   The `HR` account is unlocked and its password is *`password`*.

Now you can connect to Oracle Database as user `HR` with the password *`password`*. For instructions, see either "Connecting to Oracle Database from SQL*Plus" on page 2-1 or "Connecting to Oracle Database from SQL Developer" on page 2-2.

**See Also:**

- *Oracle Database 2 Day + Security Guide* for information about predefined user accounts provided by Oracle Database, including `HR`
- *Oracle Database SQL Developer User's Guide* for information about accessing SQL*Plus within SQL Developer
- *Oracle Database 2 Day DBA* for information about using Enterprise Manager to unlock user accounts

## Connecting to Oracle Database as User HR from SQL*Plus

This section shows how to connect to Oracle Database as the user `HR` from SQL*Plus, if the `HR` account is unlocked.

**To connect to Oracle Database as user HR from SQL*Plus:**

**Note:** For this task, you need the password for the `HR` account.

1. If you are connected to Oracle Database, close your current connection.
2. Follow the directions in "Connecting to Oracle Database from SQL*Plus" on page 2-1, entering the user name `HR` at step 3 and the password for the `HR` account at step 4.

   You are now connected to Oracle Database as the user `HR`.

**See Also:** *SQL*Plus User's Guide and Reference* for an example of using SQL*Plus to create an `HR` connection

## Connecting to Oracle Database as User HR from SQL Developer

This section shows how to connect to Oracle Database as the user HR from SQL Developer, if the HR account is unlocked.

> **Note:** For the following procedure, you need the password for the HR account.

**To connect to Oracle Database as user HR from SQL Developer:**

- Follow the directions in "Connecting to Oracle Database from SQL Developer" on page 2-2, entering the following values at steps 7 and 9.

  At step 7:

  - For Connection Name, enter `hr_conn`.

    (You can enter a different name, but the tutorials in this document assume that you named the connection `hr_conn`.)

  - For Username, enter `HR`.
  - For Password, enter the password for the `HR` account.

  At step 9:

  - For Role, accept the default.
  - For Connection Type, accept the default (Basic).
  - For Hostname, enter `localhost`.
  - For Port, enter `1521`.
  - For SID, enter `orcl`.

  At step 11, the name of the connection, `hr_conn`, appears in the Connections pane of the Oracle SQL Developer window.

You are now connected to Oracle Database as the user HR.

# 3

# Exploring Oracle Database with SQL Developer

This chapter contains the following topics:



> **Note:** To do the tutorials in this document, you must be connected to Oracle Database as the user `HR` from SQL Developer. For instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6.

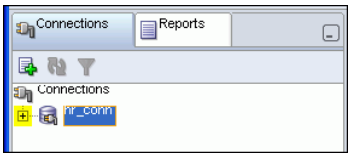
## Tutorial: Viewing HR Schema Objects

This tutorial shows how to use SQL Developer to view the objects that belong to the `HR` schema. This is called **browsing** the `HR` schema.

### To browse the HR schema:

1. In the navigation frame, click the tab **Connections**.

   The Connections pane shows the `hr_conn` icon. To the left of the icon is a plus sign (+).

2. Click the **plus sign**.

   If you are not connected to the database, the Connection Information window opens. If you are connected to the database, the `hr_conn` information expands (see the information that follows "Click OK" in step 3).

3. If the Connection Information window opens:

   1. In the User Name field, enter `hr`.
   2. In the Password field, enter the password for the user `hr`.
   3. Click **OK**.

   The `hr_conn` information expands: The plus sign becomes a minus sign, and under the `hr_conn` icon, a list of schema object types appears—Tables, Views,

Indexes, and so on. (If you click the minus sign, the hr_conn information collapses: The minus sign becomes a plus sign, and the list disappears.)

**See Also:**

- "About Schema Objects" on page 1-2 for brief descriptions of the schema object types
- *Oracle Database SQL Developer User's Guide* for more information about the SQL Developer user interface

# Tutorial: Viewing EMPLOYEES Table Properties and Data

This tutorial shows how to use SQL Developer to view the properties and data of the `EMPLOYEES` table in the `HR` schema, if you are browsing the `HR` schema.

If you are not browsing the `HR` schema, follow the instructions in "Tutorial: Viewing HR Schema Objects" on page 3-1 and then return to this topic.

**To view the properties and data of the EMPLOYEES table:**

1. In the Connections pane, expand **Tables**.

   Under Tables, a list of the tables in the `HR` schema appears.

2. Select the table **EMPLOYEES**.

   In the right frame of the Oracle SQL Developer window, in the Columns pane, a list of all columns of this table appears. To the right of each column are its properties—name, data type, and so on. (To see all column properties, move the horizontal scroll bar to the right.)

3. In the right frame, click the tab **Data**.

   The Data pane appears, showing a numbered list of all records in this table. (To see more records, move the vertical scroll bar down. To see more columns of the records, move the horizontal scroll bar to the right.)

4. In the right frame, click the tab **Constraints**.

   The Constraints pane appears, showing a list of all constraints on this table. To the right of each constraint are its properties—name, type, search condition, and so on. (To see all constraint properties, move the horizontal scroll bar to the right.)

5. Explore the other properties by clicking on the appropriate tabs.

**See Also:**

- *Oracle Database SQL Developer User's Guide* for more information about the SQL Developer user interface
- Chapter 4, "Selecting Table Data," for information about using queries to see table data

# 4

# Selecting Table Data

This chapter contains the following topics:



> **Note:** To do the tutorials in this document, you must be connected to Oracle Database as the user `HR` from SQL Developer. For instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6.

## About Queries

A **query**, or SQL `SELECT` statement, selects data from one or more tables or views.

The simplest form of query has this syntax:

```
SELECT select_list FROM source_list
```

The *`select_list`* specifies the columns from which the data is to be selected, and the *`source_list`* specifies the tables or views that have these columns.

A query nested within another SQL statement is called a **subquery**.

In the SQL*Plus environment, you can enter a query after the `SQL>` prompt.

In the SQL Developer environment, you can enter a query in the SQL Worksheet. For instructions, see "Running Queries in SQL Developer" on page 4-2.

**Note:** When the result of a query is displayed, records can be in any order, unless you specify their order with the `ORDER BY` clause. For more information, see "Sorting Selected Data" on page 4-11.

**See Also:**

- *Oracle Database SQL Language Reference* for more information about queries and subqueries
- *Oracle Database SQL Language Reference* for more information about the `SELECT` statement
- *SQL*Plus User's Guide and Reference* for more information about the SQL*Plus command line interface
- *Oracle Database SQL Developer User's Guide* for information about using the SQL Worksheet in SQL Developer

## Running Queries in SQL Developer

This topic explains how to run queries or other SQL statements in SQL Developer, if you are connected to Oracle Database as user `HR` from SQL Developer (for instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6).

**To run queries in SQL Developer:**

1. Click the icon **SQL Worksheet**.

   Either the Connection Information window opens or the SQL Worksheet pane appears.

2. If the Select Connection window opens:

   1. If the Connection field does not have the value `hr_conn`, select that value from the drop-down list.
   2. Click **OK**.

   The SQL Worksheet pane appears. Under "Enter SQL Statement:" is a field where you can enter a SQL statement.

3. In the SQL Worksheet pane, type a query (a `SELECT` statement).
4. Click the icon **Execute Statement**.

   The query runs.

5. Click the tab **Results**.

   The Results pane appears, showing the result of the query.

6. Click the icon **Clear**.

   The query and its results disappear, and you can enter another SQL statement in the SQL Worksheet.

**See Also:** *Oracle Database SQL Developer User's Guide* for information about using the SQL Worksheet in SQL Developer

# Tutorial: Selecting All Columns of a Table

This tutorial shows how to select all columns of the EMPLOYEES table.

**To select all columns of the EMPLOYEES Table:**

1. Click the icon **SQL Worksheet**.

   The SQL Worksheet pane appears.

2. In the field under "Enter SQL Statement:", enter this query:

```
SELECT * FROM EMPLOYEES;
```

3. Click the **Execute Statement**.

   The query runs.

4. Click the tab **Results**.

   The Results pane appears, showing the result of the query.

> **See Also:** "Tutorial: Viewing EMPLOYEES Table Properties and Data" on page 3-2 for information about another way to view table data with SQL Developer

# Tutorial: Selecting Specific Columns of a Table

This tutorial shows how to select only the columns FIRST_NAME, LAST_NAME, and DEPARTMENT_ID of the EMPLOYEES table.

**To select only FIRST_NAME, LAST_NAME, and DEPARTMENT_ID:**

1. Click the icon **SQL Worksheet**.

   The SQL Worksheet pane appears.

2. In the field under "Enter SQL Statement:", enter this query:

```
SELECT FIRST_NAME, LAST_NAME, DEPARTMENT_ID
FROM EMPLOYEES;
```

3. Click the icon **Execute Statement**.

   The query runs.

4. Click the tab **Results**.

   The Results pane appears, showing the results of the query, which are similar to:

```
FIRST_NAME           LAST_NAME                 DEPARTMENT_ID
-------------------- ------------------------- -------------
Donald               OConnell                             50
Douglas              Grant                                50
Jennifer             Whalen                               10
Michael              Hartstein                            20
Pat                  Fay                                  20
Susan                Mavris                               40
Hermann              Baer                                 70
Shelley              Higgins                             110
William              Gietz                               110
Steven               King                                 90
Neena                Kochhar                              90

FIRST_NAME           LAST_NAME                 DEPARTMENT_ID
```

```
------------------- ------------------------ -------------
Lex                 De Haan                            90
...
Kevin               Feeney                             50

107 rows selected.
```

# Displaying Selected Columns Under New Headings

When query results are displayed, the default column heading is the column name. To display a column under a new heading, specify the new heading (**alias**) immediately after the name of the column. The alias renames the column for the duration of the query, but does not change its name in the database.

The query in Example 4–1 selects the same columns as the query in Tutorial: Selecting Specific Columns of a Table, but it also specifies aliases for them. Because the aliases are not enclosed in double quotation marks, they are displayed in uppercase letters.

***Example 4–1 Displaying Selected Columns Under New Headings***

```
SELECT FIRST_NAME First, LAST_NAME last, DEPARTMENT_ID DepT
FROM EMPLOYEES;
```

Result is similar to:

```
FIRST               LAST                         DEPT
------------------- ------------------------ ----------
Donald              OConnell                         50
Douglas             Grant                            50
Jennifer            Whalen                           10
Michael             Hartstein                        20
Pat                 Fay                              20
Susan               Mavris                           40
Hermann             Baer                             70
Shelley             Higgins                         110
William             Gietz                           110
Steven              King                             90
Neena               Kochhar                          90

FIRST               LAST                         DEPT
------------------- ------------------------ ----------
Lex                 De Haan                          90
...
Kevin               Feeney                           50

107 rows selected.
```

If you enclose column aliases in double quotation marks, case is preserved, and the aliases can include spaces, as in Example 4–2.

***Example 4–2 Preserving Case and Including Spaces in Column Aliases***

```
SELECT FIRST_NAME "Given Name", LAST_NAME "Family Name"
FROM EMPLOYEES;
```

Result is similar to:

```
Given Name          Family Name
------------------- ------------------------
Donald              OConnell
```

```
Douglas              Grant
Jennifer             Whalen
Michael              Hartstein
Pat                  Fay
Susan                Mavris
Hermann              Baer
Shelley              Higgins
William              Gietz
Steven               King
Neena                Kochhar

Given Name           Family Name
-------------------- -------------------------
Lex                  De Haan
...
Kevin                Feeney

107 rows selected.
```

> **See Also:** *Oracle Database SQL Language Reference* for more information about the `SELECT` statement, including the column alias (*`c_alias`*)

## Selecting Data that Satisfies Specified Conditions

To select only data that matches a specified condition, include the `WHERE` clause in the `SELECT` statement. The condition in the `WHERE` clause can be any SQL condition (for information about SQL conditions, see *Oracle Database SQL Language Reference*).

The query in Example 4–3 selects data only for employees in department 90.

***Example 4–3 Selecting Data from One Department***

```
SELECT FIRST_NAME, LAST_NAME, DEPARTMENT_ID
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 90;
```

Result is similar to:

```
FIRST_NAME           LAST_NAME                 DEPARTMENT_ID
-------------------- ------------------------- -------------
Steven               King                               90
Neena                Kochhar                            90
Lex                  De Haan                            90
```

The query in Example 4–4 selects data only for employees in departments 100, 110, and 120.

***Example 4–4 Selecting Data from Specified Departments***

```
SELECT FIRST_NAME, LAST_NAME, DEPARTMENT_ID
FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (100, 110, 120);
```

Result is similar to:

```
FIRST_NAME           LAST_NAME                 DEPARTMENT_ID
-------------------- ------------------------- -------------
John                 Chen                              100
Daniel               Faviet                            100
```

```
William              Gietz                          110
Nancy                Greenberg                      100
Shelley              Higgins                        110
Luis                 Popp                           100
Ismael               Sciarra                        100
Jose Manuel          Urman                          100

8 rows selected.
```

There are no employees in department 120.

The query in Example 4–5 selects data only for employees whose last names start with "Ma".

***Example 4–5 Selecting Data for Last Names that Start with the Same Substring***

```
SELECT FIRST_NAME, LAST_NAME
FROM EMPLOYEES
WHERE LAST_NAME LIKE 'Ma%';
```

Result is similar to:

```
FIRST_NAME           LAST_NAME
-------------------- -------------------------
Jason                Mallin
Steven               Markle
James                Marlow
Mattea               Marvins
Randall              Matos
Susan                Mavris

6 rows selected.
```

The query in Example 4–6 selects data only for employees whose last names include "ma".

***Example 4–6 Selecting Data for Last Names that Include the Same Substring***

```
SELECT FIRST_NAME, LAST_NAME
FROM EMPLOYEES
WHERE LAST_NAME LIKE '%ma%';
```

Result is similar to:

```
FIRST_NAME           LAST_NAME
-------------------- -------------------------
Sundita              Kumar
Jose Manuel          Urman
Shanta               Vollman
```

The query in Example 4–7 tests for two conditions—whether the salary is at least 11000, and whether the commission percentage is not null.

***Example 4–7 Selecting Data that Satisfies Two Conditions***

```
SELECT FIRST_NAME, LAST_NAME, SALARY, COMMISSION_PCT "%"
FROM EMPLOYEES
WHERE (SALARY >= 11000) AND (COMMISSION_PCT IS NOT NULL);
```

Result is similar to:

```
FIRST_NAME           LAST_NAME                     SALARY          %
```

```
------------------- ------------------------ ---------- ----------
John                Russell                       14000         .4
Karen               Partners                      13500         .3
Alberto             Errazuriz                     12000         .3
Gerald              Cambrault                     11000         .3
Lisa                Ozer                          11500        .25
Ellen               Abel                          11000         .3

6 rows selected.
```

**See Also:**

- *Oracle Database SQL Language Reference* for more information about the `SELECT` statement, including the `WHERE` clause
- *Oracle Database SQL Language Reference* for more information about SQL conditions

## Specifying Conditions with Regular Expressions

As stated in "Selecting Data that Satisfies Specified Conditions" on page 4-5, the condition in the `WHERE` clause can be any SQL condition. This topic shows how to specify conditions with SQL functions that accept regular expressions. A regular expression defines a search pattern, using metacharacters to specify search algorithms and literals to specify characters.

Suppose that you want to select all managers in the `EMPLOYEES` table. The `JOB_ID` of a manager ends with either `'_MGR'` or `'_MAN'`, depending on the department. Therefore, the search pattern must be a regular expression, and you must use the `REGEXP_LIKE` function, as in Example 4–8.

In the regular expression `(_m[an|gr])`, the metacharacter `|` indicates the `OR` condition. The third function parameter, `'i'`, specifies that the match is case-insensitive.

***Example 4–8 Selecting All Managers in the EMPLOYEES Table***

```
SELECT FIRST_NAME, LAST_NAME, JOB_ID
FROM EMPLOYEES
WHERE REGEXP_LIKE(JOB_ID, '(_m[an|gr])', 'i');
```

Result is similar to:

```
FIRST_NAME          LAST_NAME                JOB_ID
------------------- ------------------------ ----------
Michael             Hartstein                MK_MAN
Shelley             Higgins                  AC_MGR
Nancy               Greenberg                FI_MGR
Den                 Raphaely                 PU_MAN
Matthew             Weiss                    ST_MAN
Adam                Fripp                    ST_MAN
Payam               Kaufling                 ST_MAN
Shanta              Vollman                  ST_MAN
Kevin               Mourgos                  ST_MAN
John                Russell                  SA_MAN
Karen               Partners                 SA_MAN

FIRST_NAME          LAST_NAME                JOB_ID
------------------- ------------------------ ----------
Alberto             Errazuriz                SA_MAN
Gerald              Cambrault                SA_MAN
```

```
Eleni                Zlotkey                   SA_MAN

14 rows selected.
```

Suppose that you want to select every employee whose last name has a double vowel (two adjacent occurrences of the same vowel). Example 4–9 shows how you can do this.

The regular expression `([AEIOU])` represents any vowel. The metacharacter `\1` represents the first (and in this case, only) regular expression. The third function parameter, `'i'`, specifies that the match is case-insensitive.

***Example 4–9 Selecting All Employees Whose Last Names Have Double Vowels***

```
SELECT FIRST_NAME, LAST_NAME
FROM EMPLOYEES
WHERE REGEXP_LIKE(LAST_NAME, '([AEIOU])\1', 'i');
```

Result is similar to:

```
FIRST_NAME           LAST_NAME
-------------------- -------------------------
Harrison             Bloom
Lex                  De Haan
Kevin                Feeney
Ki                   Gee
Nancy                Greenberg
Danielle             Greene
Alexander            Khoo
David                Lee

8 rows selected.
```

Suppose that, in the displayed query results, you want to replace phone numbers that are stored in the format *nnn.nnn.nnnn* with their equivalents in the format *(nnn) nnn-nnnn*. You can use the `REGEXP_REPLACE` function, with regular expressions in the search pattern (the stored format) and references to those regular expressions in the replace string (the display format), as in Example 4–10.

The search pattern has three regular expressions, each of which is enclosed in parentheses. The metacharacter `[[:digit:]]` represents a digit, the metacharacter `{n}` specifies *n* occurrences, and the metacharacter `\` is an escape character. The character immediately after an escape character is interpreted as a literal. Without the escape character, the metacharacter `.` represents any character.

The replace string uses `\1`, `\2`, and `\3` to represent the first, second, and third regular expressions in the search pattern, respectively. (In the replace string, `\` is not an escape character.)

***Example 4–10 Displaying Phone Numbers in a Different Format***

```
SELECT PHONE_NUMBER "Old Format",
REGEXP_REPLACE(PHONE_NUMBER,
'([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{4})',
'(\1) \2-\3') "New Format"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 90;
```

Result is similar to:

```
Old Format          New Format
```

```
------------------- ------------------------------------------------------
515.123.4567        (515) 123-4567
515.123.4568        (515) 123-4568
515.123.4569        (515) 123-4569
```

Suppose that you want to extract the street number from each STREET_ADDRESS in the LOCATIONS table. Some street numbers include hyphens, so the search pattern must use a regular expression, and you must use the REGEXP_SUBSTR function, as in Example 4–11.

***Example 4–11 Extracting the Street Number from Each STREET_ADDRESS***

```
COLUMN Number FORMAT A8
SELECT STREET_ADDRESS "Address",
REGEXP_SUBSTR(STREET_ADDRESS, '[[:digit:]-]+') "Number"
FROM LOCATIONS;
COLUMN Number CLEAR
```

Result is similar to:

```
Address                                Number
-------------------------------------- --------
1297 Via Cola di Rie                   1297
93091 Calle della Testa                93091
2017 Shinjuku-ku                       2017
9450 Kamiya-cho                        9450
2014 Jabberwocky Rd                    2014
2011 Interiors Blvd                    2011
2007 Zagora St                         2007
2004 Charade Rd                        2004
147 Spadina Ave                        147
6092 Boxwood St                        6092
40-5-12 Laogianggen                    40-5-12

Address                                Number
-------------------------------------- --------
1298 Vileparle (E)                     1298
12-98 Victoria Street                  12-98
198 Clementi North                     198
8204 Arthur St                         8204
Magdalen Centre, The Oxford Science Park
9702 Chester Road                      9702
Schwanthalerstr. 7031                  7031
Rua Frei Caneca 1360                   1360
20 Rue des Corps-Saints                20
Murtenstrasse 921                      921
Pieter Breughelstraat 837              837

Address                                Number
-------------------------------------- --------
Mariano Escobedo 9991                  9991

23 rows selected.
```

To count the number of spaces in each STREET_ADDRESS, you can use the REGEXP_COUNT function, as in Example 4–12.

***Example 4–12 Counting the Number of Spaces in Each STREET_ADDRESS***

```
SELECT STREET_ADDRESS,
```

```
REGEXP_COUNT(STREET_ADDRESS, ' ') "Number of Spaces"
FROM LOCATIONS;
```

Result is similar to:

```
STREET_ADDRESS                          Number of Spaces
--------------------------------------- ----------------
1297 Via Cola di Rie                                   4
93091 Calle della Testa                                3
2017 Shinjuku-ku                                       1
9450 Kamiya-cho                                        1
2014 Jabberwocky Rd                                    2
2011 Interiors Blvd                                    2
2007 Zagora St                                         2
2004 Charade Rd                                        2
147 Spadina Ave                                        2
6092 Boxwood St                                        2
40-5-12 Laogianggen                                    1

STREET_ADDRESS                          Number of Spaces
--------------------------------------- ----------------
1298 Vileparle (E)                                     2
12-98 Victoria Street                                  2
198 Clementi North                                     2
8204 Arthur St                                         2
Magdalen Centre, The Oxford Science Park               5
9702 Chester Road                                      2
Schwanthalerstr. 7031                                  1
Rua Frei Caneca 1360                                   4
20 Rue des Corps-Saints                                3
Murtenstrasse 921                                      1
Pieter Breughelstraat 837                              2

STREET_ADDRESS                          Number of Spaces
--------------------------------------- ----------------
Mariano Escobedo 9991                                  2

23 rows selected.
```

To report the position of the first space in each `STREET_ADDRESS`, you can use the `REGEXP_INSTR` function, as in Example 4–13.

***Example 4–13 Reporting the Position of the First Space in Each STREET_ADDRESS***

```
SELECT STREET_ADDRESS,
REGEXP_INSTR(STREET_ADDRESS, ' ') "First Space"
FROM LOCATIONS;
```

Result is similar to:

```
STREET_ADDRESS                          First Space
--------------------------------------- -----------
1297 Via Cola di Rie                              5
93091 Calle della Testa                           6
2017 Shinjuku-ku                                  5
9450 Kamiya-cho                                   5
2014 Jabberwocky Rd                               5
2011 Interiors Blvd                               5
2007 Zagora St                                    5
2004 Charade Rd                                   5
147 Spadina Ave                                   4
```

```
6092 Boxwood St                                    5
40-5-12 Laogianggen                                8

STREET_ADDRESS                            First Space
---------------------------------------- -----------
1298 Vileparle (E)                                 5
12-98 Victoria Street                              6
198 Clementi North                                 4
8204 Arthur St                                     5
Magdalen Centre, The Oxford Science Park           9
9702 Chester Road                                  5
Schwanthalerstr. 7031                             17
Rua Frei Caneca 1360                               4
20 Rue des Corps-Saints                            3
Murtenstrasse 921                                 14
Pieter Breughelstraat 837                          7

STREET_ADDRESS                            First Space
---------------------------------------- -----------
Mariano Escobedo 9991                              8

23 rows selected.
```

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for more information about using regular expressions in database applications
- *Oracle Database SQL Language Reference* for syntax of regular expressions
- *Oracle Database SQL Language Reference* for more information about the `REGEXP_LIKE` expression
- *Oracle Database SQL Language Reference* for more information about the `REGEXP_REPLACE` expression
- *Oracle Database SQL Language Reference* for more information about the `REGEXP_SUBSTR` expression
- *Oracle Database SQL Language Reference* for more information about the `REGEXP_COUNT` expression
- *Oracle Database SQL Language Reference* for more information about the `REGEXP_INSTR` expression

## Sorting Selected Data

When the results of a query are displayed, records can be in any order, unless you specify their order with the `ORDER BY` clause.

The results of the query in Example 4–14 are sorted by `LAST_NAME`, in ascending order (the default).

Alternatively, in SQL Developer, you can omit the `ORDER BY` clause and double-click the name of the column to sort.

***Example 4–14 Sorting Selected Data by LAST_NAME***

```
SELECT FIRST_NAME, LAST_NAME, HIRE_DATE
FROM EMPLOYEES
```

```
ORDER BY LAST_NAME;
```

Result:

```
FIRST_NAME           LAST_NAME                 HIRE_DATE
-------------------- ------------------------- ---------
Ellen                Abel                      11-MAY-96
Sundar               Ande                      24-MAR-00
Mozhe                Atkinson                  30-OCT-97
David                Austin                    25-JUN-97
Hermann              Baer                      07-JUN-94
Shelli               Baida                     24-DEC-97
Amit                 Banda                     21-APR-00
Elizabeth            Bates                     24-MAR-99
...
FIRST_NAME           LAST_NAME                 HIRE_DATE
-------------------- ------------------------- ---------
Jose Manuel          Urman                     07-MAR-98
Peter                Vargas                    09-JUL-98
Clara                Vishney                   11-NOV-97
Shanta               Vollman                   10-OCT-97
Alana                Walsh                     24-APR-98
Matthew              Weiss                     18-JUL-96
Jennifer             Whalen                    17-SEP-87
Eleni                Zlotkey                   29-JAN-00

107 rows selected
```

The sort criterion need not be included in the select list, as Example 4–15 shows.

***Example 4–15 Sorting Selected Data by an Unselected Column***

```
SELECT FIRST_NAME, HIRE_DATE
FROM EMPLOYEES
ORDER BY LAST_NAME;
```

Result:

```
FIRST_NAME           HIRE_DATE
-------------------- ---------
Ellen                11-MAY-96
Sundar               24-MAR-00
Mozhe                30-OCT-97
David                25-JUN-97
Hermann              07-JUN-94
Shelli               24-DEC-97
Amit                 21-APR-00
Elizabeth            24-MAR-99
...
FIRST_NAME           HIRE_DATE
-------------------- ---------
Jose Manuel          07-MAR-98
Peter                09-JUL-98
Clara                11-NOV-97
Shanta               10-OCT-97
Alana                24-APR-98
Matthew              18-JUL-96
Jennifer             17-SEP-87
Eleni                29-JAN-00

107 rows selected.
```

**See Also:** *Oracle Database SQL Language Reference* for more information about the SELECT statement, including the ORDER BY clause

## Selecting Data from Multiple Tables

Suppose that you want to select the FIRST_NAME, LAST_NAME, and DEPARTMENT_NAME of every employee. FIRST_NAME and LAST_NAME are in the EMPLOYEES table, and DEPARTMENT_NAME is in the DEPARTMENTS table. Both tables have DEPARTMENT_ID. You can use the query in Example 4–16. Such a query is called a **join**.

***Example 4–16 Selecting Data from Two Tables (Joining Two Tables)***

```
SELECT EMPLOYEES.FIRST_NAME "First",
EMPLOYEES.LAST_NAME "Last",
DEPARTMENTS.DEPARTMENT_NAME "Dept. Name"
FROM EMPLOYEES, DEPARTMENTS
WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID;
```

Result:

```
First                Last                      Dept. Name
-------------------- ------------------------- ------------------------------
Jennifer             Whalen                    Administration
Michael              Hartstein                 Marketing
Pat                  Fay                       Marketing
Den                  Raphaely                  Purchasing
Karen                Colmenares                Purchasing
Alexander            Khoo                      Purchasing
Shelli               Baida                     Purchasing
Sigal                Tobias                    Purchasing
Guy                  Himuro                    Purchasing
Susan                Mavris                    Human Resources
Donald               OConnell                  Shipping

First                Last                      Dept. Name
-------------------- ------------------------- ------------------------------
Douglas              Grant                     Shipping
...
Shelley              Higgins                   Accounting

106 rows selected.
```

Table-name qualifiers are optional for column names that appear in only one table of a join, but are required for column names that appear in both tables. The following query is equivalent to the query in Example 4–16:

```
SELECT FIRST_NAME "First",
LAST_NAME "Last",
DEPARTMENT_NAME "Dept. Name"
FROM EMPLOYEES, DEPARTMENTS
WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID;
```

To make queries that use qualified column names more readable, use table aliases, as in the following example:

```
SELECT FIRST_NAME "First",
LAST_NAME "Last",
DEPARTMENT_NAME "Dept. Name"
```

```
FROM EMPLOYEES e, DEPARTMENTS d
WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID;
```

Although you create the aliases in the `FROM` clause, you can use them earlier in the query, as in the following example:

```
SELECT e.FIRST_NAME "First",
e.LAST_NAME "Last",
d.DEPARTMENT_NAME "Dept. Name"
FROM EMPLOYEES e, DEPARTMENTS d
WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID;
```

**See Also:** *Oracle Database SQL Language Reference* for more information about joins

# Using Operators and Functions in Queries

The *`select_list`* of a query can include SQL expressions, which can include SQL operators and SQL functions. These operators and functions can have table data as operands and arguments. The SQL expressions are evaluated, and their values appear in the results of the query.

Topics:

- Using Arithmetic Operators in Queries
- Using Numeric Functions in Queries
- Using the Concatenation Operator in Queries
- Using Character Functions in Queries
- Using Datetime Functions in Queries
- Using Conversion Functions in Queries
- Using Aggregate Functions in Queries
- Using NULL-Related Functions in Queries
- Using CASE Expressions in Queries
- Using the DECODE Function in Queries

**See Also:**

- *Oracle Database SQL Language Reference* for more information about SQL operators
- *Oracle Database SQL Language Reference* for more information about SQL functions

## Using Arithmetic Operators in Queries

SQL supports the basic arithmetic operators: + (addition), – (subtraction), * (multiplication), and / (division).

The query in Example 4–17 displays `LAST_NAME`, `SALARY` (monthly pay), and annual pay for each employee in department 90, in descending order of `SALARY`.

***Example 4–17 Using an Arithmetic Expression in a Query***

```
SELECT LAST_NAME,
SALARY "Monthly Pay",
```

```
SALARY * 12 "Annual Pay"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 90
ORDER BY SALARY DESC;
```

Result:

```
LAST_NAME                 Monthly Pay Annual Pay
------------------------- ----------- ----------
King                            24000     288000
De Haan                         17000     204000
Kochhar                         17000     204000
```

## Using Numeric Functions in Queries

Numeric functions accept numeric input and return numeric values. Each numeric function returns a single value for each row that is evaluated. The numeric functions that SQL supports are listed and described in *Oracle Database SQL Language Reference*.

The query in Example 4–18 uses the numeric function ROUND to display the daily pay of each employee in department 100, rounded to the nearest cent.

***Example 4–18 Rounding Numeric Data***

```
SELECT LAST_NAME,
ROUND (((SALARY * 12)/365), 2) "Daily Pay"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                  Daily Pay
------------------------- ----------
Chen                          269.59
Faviet                        295.89
Greenberg                     394.52
Popp                          226.85
Sciarra                       253.15
Urman                         256.44

6 rows selected.
```

The query in Example 4–19 uses the numeric function TRUNC to display the daily pay of each employee in department 100, truncated to the nearest dollar.

***Example 4–19 Truncating Numeric Data***

```
SELECT LAST_NAME,
TRUNC (((SALARY * 12)/365), 2) "Daily Pay"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                  Daily Pay
------------------------- ----------
Chen                             269
Faviet                           295
Greenberg                        394
```

```
Popp                           226
Sciarra                        253
Urman                          256

6 rows selected.
```

> **See Also:** *Oracle Database SQL Language Reference* for more information about SQL numeric functions

## Using the Concatenation Operator in Queries

The concatenation operator (||) combines two strings into one string, by appending the second string to the first. For example, `'a'||'b'='ab'`. You can use this operator to combine information from two columns or expressions in the same column of the report, as in the query in Example 4–20.

***Example 4–20 Concatenating Character Data***

```
SELECT FIRST_NAME || ' ' || LAST_NAME "Name"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
Name
---------------------------------------------
John Chen
Daniel Faviet
Nancy Greenberg
Luis Popp
Ismael Sciarra
Jose Manuel Urman

6 rows selected.
```

## Using Character Functions in Queries

Character functions accept character input. Most return character values, but some return numeric values. Each character function returns a single value for each row that is evaluated. The character functions that SQL supports are listed and described in *Oracle Database SQL Language Reference*.

The functions `UPPER`, `INITCAP`, and `LOWER` display their character arguments in uppercase, initial capital, and lowercase, respectively.

The query in Example 4–21 displays `LAST_NAME` in uppercase, `FIRST_NAME` with the first character in uppercase and all others in lowercase, and `EMAIL` in lowercase.

***Example 4–21 Changing the Case of Character Data***

```
SELECT UPPER(LAST_NAME) "Last",
INITCAP(FIRST_NAME) "First",
LOWER(EMAIL) "E-Mail"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY EMAIL;
```

Result:

```
Last                      First               E-Mail
```

```
------------------------ ------------------- ------------------------
FAVIET                   Daniel              dfaviet
SCIARRA                  Ismael              isciarra
CHEN                     John                jchen
URMAN                    Jose Manuel         jmurman
POPP                     Luis                lpopp
GREENBERG                Nancy               ngreenbe

6 rows selected.
```

The functions `LTRIM` and `RTRIM` trim characters from the left and right ends of their character arguments, respectively. The function `TRIM` trims leading zeros, trailing zeros, or both.

The query in Example 4–22 finds every clerk in the `EMPLOYEES` table and trims `'_CLERK'` from the `JOB_ID`, displaying only the characters that identify the type of clerk.

***Example 4–22 Trimming Character Data***

```
SELECT LAST_NAME,
RTRIM(JOB_ID, '_CLERK') "Clerk Type"
FROM EMPLOYEES
WHERE JOB_ID LIKE '%_CLERK'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                Clerk Type
------------------------ ----------
Atkinson                 ST
Baida                    PU
Bell                     SH
Bissot                   ST
Bull                     SH
Cabrio                   SH
Chung                    SH
Colmenares               PU
Davies                   ST
Dellinger                SH
Dilly                    SH

LAST_NAME                Clerk Type
------------------------ ----------
Everett                  SH
Feeney                   SH
...
LAST_NAME                Clerk Type
------------------------ ----------
Walsh                    SH

45 rows selected.
```

The functions `LPAD` and `RPAD` pad their character arguments on the left and right, respectively, with a specified character (the default character is a space).

The query in Example 4–23 displays `FIRST_NAME` and `LAST_NAME` in 15-character columns, blank-padding `FIRST_NAME` on the left and `LAST_NAME` on the right.

***Example 4–23 Padding Character Data***

```
SELECT LPAD(FIRST_NAME,15) "First",
RPAD(LAST_NAME,15) "Last"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY FIRST_NAME;
```

Result:

```
First           Last
--------------- ---------------
         Daniel Faviet
         Ismael Sciarra
           John Chen
    Jose Manuel Urman
           Luis Popp
          Nancy Greenberg

6 rows selected.
```

The SUBSTR function accepts as arguments a string, a character position, and a length, and returns the substring that starts at the specified position in the string and has the specified length.

The query in Example 4–24 uses SUBSTR to abbreviate FIRST_NAME to first initial and to strip the area code from PHONE_NUMBER.

***Example 4–24 Extracting Substrings from Character Data***

```
SELECT SUBSTR(FIRST_NAME, 1, 1) || '. ' || LAST_NAME "Name",
SUBSTR(PHONE_NUMBER, 5, 8) "Phone"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
Name                        Phone
--------------------------- --------
J. Chen                     124.4269
D. Faviet                   124.4169
N. Greenberg                124.4569
L. Popp                     124.4567
I. Sciarra                  124.4369
J. Urman                    124.4469

6 rows selected.
```

The REPLACE function replaces one substring with another.

The query in Example 4–25 uses the SUBSTR function in the WHERE clause to select the employees whose JOB_ID starts with 'SH', and uses the REPLACE function to replace 'SH' with 'SHIPPING' in each such JOB_ID.

***Example 4–25 Replacing Substrings in Character Data***

```
COLUMN "Job" FORMAT A15;
SELECT LAST_NAME,
REPLACE(JOB_ID, 'SH', 'SHIPPING') "Job"
FROM EMPLOYEES
WHERE SUBSTR(JOB_ID, 1, 2) = 'SH'
```

```
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 Job
------------------------- ---------------
Bell                      SHIPPING_CLERK
Bull                      SHIPPING_CLERK
Cabrio                    SHIPPING_CLERK
Chung                     SHIPPING_CLERK
Dellinger                 SHIPPING_CLERK
Dilly                     SHIPPING_CLERK
Everett                   SHIPPING_CLERK
Feeney                    SHIPPING_CLERK
Fleaur                    SHIPPING_CLERK
Gates                     SHIPPING_CLERK
Geoni                     SHIPPING_CLERK

LAST_NAME                 Job
------------------------- ---------------
Grant                     SHIPPING_CLERK
Jones                     SHIPPING_CLERK
McCain                    SHIPPING_CLERK
OConnell                  SHIPPING_CLERK
Perkins                   SHIPPING_CLERK
Sarchand                  SHIPPING_CLERK
Sullivan                  SHIPPING_CLERK
Taylor                    SHIPPING_CLERK
Walsh                     SHIPPING_CLERK

20 rows selected.
```

**See Also:** *Oracle Database SQL Language Reference* for more information about SQL character functions

## Using Datetime Functions in Queries

Datetime functions operate on date, timestamp, and interval values. Each datetime function returns a single value for each row that is evaluated. The datetime functions that SQL supports are listed and described in *Oracle Database SQL Language Reference*.

To understand Example 4–26, you must understand the `JOB_HISTORY` table.

When an employee changes jobs, the `START_DATE` and `END_DATE` of his or her previous job are recorded in the `JOB_HISTORY` table. Employees who have changed jobs more than once have multiple rows in the `JOB_HISTORY` table, as the following query and its results show:

```
SELECT * FROM JOB_HISTORY
ORDER BY EMPLOYEE_ID;
```

Result:

```
EMPLOYEE_ID START_DAT END_DATE  JOB_ID     DEPARTMENT_ID
----------- --------- --------- ---------- -------------
        101 21-SEP-89 27-OCT-93 AC_ACCOUNT           110
        101 28-OCT-93 15-MAR-97 AC_MGR               110
        102 13-JAN-93 24-JUL-98 IT_PROG               60
        114 24-MAR-98 31-DEC-99 ST_CLERK              50
        122 01-JAN-99 31-DEC-99 ST_CLERK              50
        176 24-MAR-98 31-DEC-98 SA_REP                80
```

```
       176 01-JAN-99 31-DEC-99 SA_MAN                80
       200 17-SEP-87 17-JUN-93 AD_ASST               90
       200 01-JUL-94 31-DEC-98 AC_ACCOUNT            90
       201 17-FEB-96 19-DEC-99 MK_REP                20

10 rows selected.
```

The query in Example 4–26 uses the MONTHS_BETWEEN function to show how many months each employee held each of his or her previous jobs. For information about the MONTHS_BETWEEN function, see *Oracle Database SQL Language Reference*.

***Example 4–26 Displaying the Number of Months Between Dates***

```
SELECT e.EMPLOYEE_ID,
e.LAST_NAME,
TRUNC(MONTHS_BETWEEN(j.END_DATE, j.START_DATE)) "Months Worked"
FROM EMPLOYEES e, JOB_HISTORY j
WHERE e.EMPLOYEE_ID = j.EMPLOYEE_ID
ORDER BY "Months Worked";
```

Result:

```
EMPLOYEE_ID LAST_NAME                 Months Worked
----------- ------------------------- -------------
       176 Taylor                                 9
       122 Kaufling                              11
       176 Taylor                                11
       114 Raphaely                              21
       101 Kochhar                               40
       201 Hartstein                             46
       101 Kochhar                               49
       200 Whalen                                53
       102 De Haan                               66
       200 Whalen                                69

10 rows selected.
```

The query in Example 4–27 uses the EXTRACT and SYSDATE functions to show how many years each employee in department 100 has been employed. The SYSDATE function returns the current date of the system clock. For more information about the SYSDATE function, see *Oracle Database SQL Language Reference*. For information about the EXTRACT function, see *Oracle Database SQL Language Reference*.

***Example 4–27 Displaying the Number of Years Between Dates***

```
SELECT LAST_NAME,
(EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM HIRE_DATE)) "Years Employed"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY "Years Employed";
```

Result:

```
LAST_NAME                 Years Employed
------------------------- --------------
Popp                                   9
Urman                                 10
Chen                                  11
Sciarra                               11
Greenberg                             14
Faviet                                14
```

```
6 rows selected.
```

Suppose that an employee receives his or her first check on the last day of the month in which he or she was hired. The query in Example 4–28 uses the LAST_DAY function to show the first pay day for each employee in department 100. For information about the LAST_DAY function, see *Oracle Database SQL Language Reference*.

***Example 4–28 Displaying the Last Day of a Selected Month***

```
SELECT LAST_NAME,
HIRE_DATE "Hired",
LAST_DAY(HIRE_DATE) "Paid"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY HIRE_DATE;
```

Result:

```
LAST_NAME                 Hired     Paid
------------------------- --------- ---------
Faviet                    16-AUG-94 31-AUG-94
Greenberg                 17-AUG-94 31-AUG-94
Chen                      28-SEP-97 30-SEP-97
Sciarra                   30-SEP-97 30-SEP-97
Urman                     07-MAR-98 31-MAR-98
Popp                      07-DEC-99 31-DEC-99

6 rows selected.
```

Suppose that an employee receives his or her first evaluation six months after being hired. The query in Example 4–29 uses the ADD_MONTHS function to show the first evaluation day for each employee in department 100. For information about the ADD_MONTHS function, see *Oracle Database SQL Language Reference*.

***Example 4–29 Displaying a Date Six Months from a Selected Date***

```
SELECT LAST_NAME,
HIRE_DATE "Hired",
ADD_MONTHS(HIRE_DATE, 6) "Evaluated"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY HIRE_DATE;
```

Result:

```
LAST_NAME                 Hired     Evaluated
------------------------- --------- ---------
Faviet                    16-AUG-94 16-FEB-95
Greenberg                 17-AUG-94 17-FEB-95
Chen                      28-SEP-97 28-MAR-98
Sciarra                   30-SEP-97 31-MAR-98
Urman                     07-MAR-98 07-SEP-98
Popp                      07-DEC-99 07-JUN-00

6 rows selected.
```

The query in Example 4–30 uses the SYSTIMESTAMP function to display the current system time and date. SYSTIMESTAMP is similar to SYSDATE, but it returns more

information. For information about the SYSTIMESTAMP function, see *Oracle Database SQL Language Reference*.

The table in the FROM clause of the query, DUAL, is a one-row table that Oracle Database creates automatically along with the data dictionary. Select from DUAL when you want to compute a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once. For more information about selecting from DUAL, see *Oracle Database SQL Language Reference*.

***Example 4–30 Displaying System Date and Time***

```
SELECT EXTRACT(HOUR FROM SYSTIMESTAMP) || ':' ||
EXTRACT(MINUTE FROM SYSTIMESTAMP) || ':' ||
ROUND(EXTRACT(SECOND FROM SYSTIMESTAMP), 0) || ', ' ||
EXTRACT(MONTH FROM SYSTIMESTAMP) || '/' ||
EXTRACT(DAY FROM SYSTIMESTAMP) || '/' ||
EXTRACT(YEAR FROM SYSTIMESTAMP) "System Time and Date"
FROM DUAL;
```

Results depend on current SYSTIMESTAMP value, but have this format:

```
System Time and Date
---------------------------------------------------------------------
18:47:33, 6/19/2008
```

> **See Also:** *Oracle Database SQL Language Reference* for more information about SQL datetime functions

## Using Conversion Functions in Queries

Conversion functions convert one data type to another. The conversion functions that SQL supports are listed and described in *Oracle Database SQL Language Reference*.

The query in Example 4–31 uses the TO_CHAR function to convert HIRE_DATE values (which are of type DATE) to character values that have the format FMMonth DD YYYY. FM removes leading and trailing blanks from the month name. FMMonth DD YYYY is an example of a **datetime format model**.

***Example 4–31 Converting Dates to Characters Using a Format Template***

```
SELECT LAST_NAME,
HIRE_DATE,
TO_CHAR(HIRE_DATE, 'FMMonth DD YYYY') "Date Started"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 HIRE_DATE Date Started
------------------------- --------- ------------------
Chen                      28-SEP-97 September 28 1997
Faviet                    16-AUG-94 August 16 1994
Greenberg                 17-AUG-94 August 17 1994
Popp                      07-DEC-99 December 7 1999
Sciarra                   30-SEP-97 September 30 1997
Urman                     07-MAR-98 March 7 1998

6 rows selected.
```

The query in Example 4–32 uses the TO_CHAR function to convert HIRE_DATE values to character values that have the two standard formats DS (Date Short) and DL (Date Long).

***Example 4–32 Converting Dates to Characters Using Standard Formats***

```
SELECT LAST_NAME,
TO_CHAR(HIRE_DATE, 'DS') "Short Date",
TO_CHAR(HIRE_DATE, 'DL') "Long Date"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 Short Date Long Date
------------------------- ---------- ----------------------------
Chen                      9/28/1997  Sunday, September 28, 1997
Faviet                    8/16/1994  Tuesday, August 16, 1994
Greenberg                 8/17/1994  Wednesday, August 17, 1994
Popp                      12/7/1999  Tuesday, December 07, 1999
Sciarra                   9/30/1997  Tuesday, September 30, 1997
Urman                     3/7/1998   Saturday, March 07, 1998

6 rows selected.
```

The query in Example 4–33 uses the TO_CHAR function to convert SALARY values (which are of type NUMBER) to character values that have the format $99,999.99.

***Example 4–33 Converting Numbers to Characters Using a Format Template***

```
SELECT LAST_NAME,
TO_CHAR(SALARY, '$99,999.99') "Salary"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY SALARY;
```

Result:

```
LAST_NAME                 Salary
------------------------- -----------
Popp                        $6,900.00
Sciarra                     $7,700.00
Urman                       $7,800.00
Chen                        $8,200.00
Faviet                      $9,000.00
Greenberg                  $12,000.00

6 rows selected.
```

The query in Example 4–34 uses the TO_NUMBER function to convert POSTAL_CODE values (which are of type VARCHAR2) to values of type NUMBER, which it uses in calculations.

***Example 4–34 Converting Characters to Numbers***

```
SELECT CITY,
POSTAL_CODE "Old Code",
TO_NUMBER(POSTAL_CODE) + 1 "New Code"
FROM LOCATIONS
WHERE COUNTRY_ID = 'US'
```

```
ORDER BY POSTAL_CODE;
```

Result:

```
CITY                           Old Code       New Code
------------------------------ ------------ ----------
Southlake                      26192             26193
South Brunswick                50090             50091
Seattle                        98199             98200
South San Francisco            99236             99237

4 rows selected.
```

The query in Example 4–35 uses the `TO_DATE` function to convert a string of characters whose format is `Month dd, YYYY, HH:MI A.M.` to a `DATE` value.

***Example 4–35 Converting a Character String to a Date***

```
SELECT TO_DATE('January 5, 2007, 8:43 A.M.',
'Month dd, YYYY, HH:MI A.M.') "Date"
FROM DUAL;
```

Result:

```
Date
---------
05-JAN-07
```

The query in Example 4–36 uses the `TO_TIMESTAMP` function to convert a string of characters whose format is `DD-Mon-RR HH24:MI:SS.FF` to a `TIMESTAMP` value.

***Example 4–36 Converting a Character String to a Time Stamp***

```
SELECT TO_TIMESTAMP('May 5, 2007, 8:43 A.M.',
'Month dd, YYYY, HH:MI A.M.') "Timestamp"
FROM DUAL;
```

Result:

```
Timestamp
---------------------------------------------------------------------------
05-MAY-07 08.43.00.000000000 AM
```

**See Also:**

- *Oracle Database SQL Language Reference* for more information about SQL conversion functions
- *Oracle Database SQL Language Reference* for more information about the `TO_CHAR` function
- *Oracle Database SQL Language Reference* for more information about the `TO_NUMBER` function
- *Oracle Database SQL Language Reference* for more information about the `TO_DATE` function
- *Oracle Database SQL Language Reference* for more information about the `TO_TIMESTAMP` function
- "About the NLS_DATE_FORMAT Parameter" on page 9-14

## Using Aggregate Functions in Queries

An aggregate function returns a single result row, based on a group of rows. The group of rows can be an entire table or view. The aggregate functions that SQL supports are listed and described in *Oracle Database SQL Language Reference*.

Aggregate functions are especially powerful when used with the `GROUP BY` clause, which groups query results by one or more columns, with a result for each group.

The query in Example 4–37 uses the `COUNT` function and the `GROUP BY` clause to show how many people report to each manager. The wildcard character, `*`, represents an entire record.

***Example 4–37 Counting the Number of Rows in Each Group***

```
SELECT MANAGER_ID "Manager",
COUNT(*) "Number of Reports"
FROM EMPLOYEES
GROUP BY MANAGER_ID;
```

Result:

```
   Manager Number of Reports
---------- -----------------
       100                14
                           1
       123                 8
       120                 8
       121                 8
       147                 6
       205                 1
       108                 5
       148                 6
       149                 6
       201                 1

   Manager Number of Reports
---------- -----------------
       102                 1
       101                 5
       114                 5
       124                 8
       145                 6
       146                 6
       103                 4
       122                 8

19 rows selected.
```

Example 4–37 shows that one employee does not report to a manager. The following query selects the first name, last name, and job title of that employee:

```
COLUMN FIRST_NAME FORMAT A10;
COLUMN LAST_NAME FORMAT A10;
COLUMN JOB_TITLE FORMAT A10;

SELECT e.FIRST_NAME,
e.LAST_NAME,
j.JOB_TITLE
FROM EMPLOYEES e, JOBS j
WHERE e.JOB_ID = j.JOB_ID
AND MANAGER_ID IS NULL;
```

Result:

```
FIRST_NAME LAST_NAME  JOB_TITLE
---------- ---------- ----------
Steven     King       President
```

When used with the DISTINCT option, the COUNT function shows how many distinct values are in a data set.

The two queries in Example 4–38 show the total number of departments and the number of departments that have employees.

***Example 4–38 Counting the Number of Distinct Values in a Set***

```
SELECT COUNT(*) FROM DEPARTMENTS;
```

Result:

```
  COUNT(*)
----------
        27
```

```
SELECT COUNT(DISTINCT DEPARTMENT_ID) "Number of Departments"
FROM EMPLOYEES;
```

Result:

```
Number of Departments
---------------------
                   11
```

The query in Example 4–39 uses several aggregate functions to show statistics for the salaries of each JOB_ID.

***Example 4–39 Using Aggregate Functions for Statistical Information***

```
SELECT JOB_ID,
COUNT(*) "#",
MIN(SALARY) "Minimum",
ROUND(AVG(SALARY), 0) "Average",
MEDIAN(SALARY) "Median",
MAX(SALARY) "Maximum",
ROUND(STDDEV(SALARY)) "Std Dev"
FROM EMPLOYEES
GROUP BY JOB_ID
ORDER BY JOB_ID;
```

Result:

```
JOB_ID              #    Minimum    Average     Median    Maximum    Std Dev
---------- ---------- ---------- ---------- ---------- ---------- ----------
AC_ACCOUNT          1       8300       8300       8300       8300          0
AC_MGR              1      12000      12000      12000      12000          0
AD_ASST             1       4400       4400       4400       4400          0
AD_PRES             1      24000      24000      24000      24000          0
AD_VP               2      17000      17000      17000      17000          0
FI_ACCOUNT          5       6900       7920       7800       9000        766
FI_MGR              1      12000      12000      12000      12000          0
HR_REP              1       6500       6500       6500       6500          0
IT_PROG             5       4200       5760       4800       9000       1926
MK_MAN              1      13000      13000      13000      13000          0
```

```
MK_REP              1       6000       6000       6000       6000          0

JOB_ID              #    Minimum    Average     Median    Maximum    Std Dev
---------- ---------- ---------- ---------- ---------- ---------- ----------
PR_REP              1      10000      10000      10000      10000          0
PU_CLERK            5       2500       2780       2800       3100        239
PU_MAN              1      11000      11000      11000      11000          0
SA_MAN              5      10500      12200      12000      14000       1525
SA_REP             30       6100       8350       8200      11500       1524
SH_CLERK           20       2500       3215       3100       4200        548
ST_CLERK           20       2100       2785       2700       3600        453
ST_MAN              5       5800       7280       7900       8200       1066

19 rows selected.
```

To have the query return only rows where aggregate values meet specified conditions, use the HAVING clause.

The query in Example 4–40 shows how much each department spends annually on salaries, but only for departments for which that amount exceeds $1,000,000.

***Example 4–40 Limiting Aggregate Functions to Rows that Satisfy a Condition***

```
SELECT DEPARTMENT_ID "Department",
SUM(SALARY*12) "All Salaries"
FROM EMPLOYEES
HAVING SUM(SALARY * 12) >= 1000000
GROUP BY DEPARTMENT_ID;
```

Result:

```
Department All Salaries
---------- ------------
        50      1876800
        80      3654000
```

The RANK function returns the relative ordered rank of a number, and the PERCENT_RANK function returns the percentile position of a number.

The query in Example 4–41 shows that a salary of $3,000 is the 20th highest, and is in the 42nd percentile, among all clerks.

***Example 4–41 Showing the Rank and Percentile of a Number Within a Group***

```
SELECT RANK(3000) WITHIN GROUP
  (ORDER BY SALARY DESC) "Rank",
ROUND(100 * (PERCENT_RANK(3000) WITHIN GROUP
  (ORDER BY SALARY DESC)), 0) "Percentile"
FROM EMPLOYEES
WHERE JOB_ID LIKE '%CLERK';
```

Result:

```
      Rank Percentile
---------- ----------
        20         42
```

The DENSE_RANK function is like the RANK function, except that the identical values have the same rank, and there are no gaps in the ranking. Using the DENSE_RANK function, $3,000 is the 12th highest salary for clerks, as Example 4–42 shows.

***Example 4–42 Showing the Dense Rank of a Number Within a Group***

```
SELECT DENSE_RANK(3000) WITHIN GROUP (ORDER BY salary DESC) "Rank"
FROM EMPLOYEES
WHERE JOB_ID LIKE '%CLERK';
```

Result:

```
      Rank
----------
        12
```

**See Also:** *Oracle Database SQL Language Reference* for more informaton about SQL aggregate functions

## Using NULL-Related Functions in Queries

The NULL-related functions facilitate the handling of NULL values. The NULL-related functions that SQL supports are listed and described in *Oracle Database SQL Language Reference*.

The query in Example 4–43 returns the last name and commission of the employees whose last names begin with 'B'. If an employee receives no commission (that is, if COMMISSION_PCT is NULL), the NVL function substitutes "Not Applicable" for NULL.

***Example 4–43 Substituting a String for a NULL Value***

```
SELECT LAST_NAME,
NVL(TO_CHAR(COMMISSION_PCT), 'Not Applicable') "COMMISSION"
FROM EMPLOYEES
WHERE LAST_NAME LIKE 'B%'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 COMMISSION
------------------------- ----------------------------------------
Baer                      Not Applicable
Baida                     Not Applicable
Banda                     .1
Bates                     .15
Bell                      Not Applicable
Bernstein                 .25
Bissot                    Not Applicable
Bloom                     .2
Bull                      Not Applicable

9 rows selected.
```

The query in Example 4–44 returns the last name, salary, and income of the employees whose last names begin with 'B', using the NVL2 function: If COMMISSION_PCT is not NULL, the income is the salary plus the commission; if COMMISSION_PCT is NULL, income is only the salary.

***Example 4–44 Specifying Different Expressions for NULL and Not NULL Values***

```
SELECT LAST_NAME, SALARY,
NVL2(COMMISSION_PCT, SALARY + (SALARY * COMMISSION_PCT), SALARY) INCOME
FROM EMPLOYEES WHERE LAST_NAME LIKE 'B%'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                    SALARY     INCOME
------------------------- ---------- ----------
Baer                          10000      10000
Baida                          2900       2900
Banda                          6200       6882
Bates                          7300       8468
Bell                           4000       4000
Bernstein                      9500      11970
Bissot                         3300       3300
Bloom                         10000      12100
Bull                           4100       4100

9 rows selected.
```

**See Also:**

- *Oracle Database SQL Language Reference* for more information about the `NVL` function
- *Oracle Database SQL Language Reference* for more information about the `NVL2` function

## Using CASE Expressions in Queries

A `CASE` expression lets you use `IF` ... `THEN` ... `ELSE` logic in SQL statements without invoking procedures.

The query in Example 4–45 uses a `CASE` expression to show proposed salary increases, based on length of service.

***Example 4–45 Using a CASE Expression in a Query***

```
SELECT LAST_NAME "Name",
HIRE_DATE "Started",
SALARY "Salary",
CASE
  WHEN HIRE_DATE < TO_DATE('01-Jan-90') THEN TRUNC(SALARY*1.15, 0)
  WHEN HIRE_DATE < TO_DATE('01-Jan-95') THEN TRUNC(SALARY*1.10, 0)
  WHEN HIRE_DATE < TO_DATE('01-Jan-00') THEN TRUNC(SALARY*1.05, 0)
  ELSE SALARY
END "Proposed Salary"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY HIRE_DATE;
```

Result:

```
Name                      Started       Salary Proposed Salary
------------------------- --------- ---------- ---------------
Faviet                    16-AUG-94       9000            9900
Greenberg                 17-AUG-94      12000           13200
Chen                      28-SEP-97       8200            8610
Sciarra                   30-SEP-97       7700            8085
Urman                     07-MAR-98       7800            8190
Popp                      07-DEC-99       6900            7245

6 rows selected.
```

**See Also:**

- "Using the CASE Statement" on page 7-24
- *Oracle Database SQL Language Reference* for more information about CASE expressions

## Using the DECODE Function in Queries

The DECODE function compares a value or expression to search values, and returns a result when it finds a match. If a match is not found, then DECODE returns the default value, or NULL (if a default value is not specified).

The query in Example 4–46 uses the DECODE function to show proposed salary increases for three different jobs.

***Example 4–46 Using the DECODE Function in a Query***

```
SELECT LAST_NAME, JOB_ID, SALARY,
DECODE(JOB_ID,
  'PU_CLERK', SALARY * 1.10,
  'SH_CLERK', SALARY * 1.15,
  'ST_CLERK', SALARY * 1.20,
  SALARY) "Proposed Salary"
FROM EMPLOYEES
WHERE JOB_ID LIKE '%_CLERK'
AND LAST_NAME < 'E'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 JOB_ID         SALARY Proposed Salary
------------------------- ---------- ---------- ---------------
Atkinson                  ST_CLERK         2800            3360
Baida                     PU_CLERK         2900            3190
Bell                      SH_CLERK         4000            4600
Bissot                    ST_CLERK         3300            3960
Bull                      SH_CLERK         4100            4715
Cabrio                    SH_CLERK         3000            3450
Chung                     SH_CLERK         3800            4370
Colmenares                PU_CLERK         2500            2750
Davies                    ST_CLERK         3100            3720
Dellinger                 SH_CLERK         3400            3910
Dilly                     SH_CLERK         3600            4140

11 rows selected.
```

**See Also:** *Oracle Database SQL Language Reference* for information about the DECODE function

# 5

# About DML Statements and Transactions

This chapter contains the following topics:



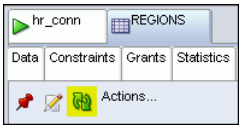
## About Data Manipulation Language (DML) Statements

**Data manipulation language (DML) statements** access and manipulate data in existing tables.

In the SQL*Plus environment, you can enter a DML statement after the `SQL>` prompt.

In the SQL Developer environment, you can enter a DML statement in the SQL Worksheet. Alternatively, you can use the SQL Developer navigation frame and tools to access and manipulate data.

To see the effect of a DML statement in SQL Developer, you might have to click the Refresh icon.




The effect of a DML statement is not permanent until you commit the transaction that includes it. A **transaction** is a sequence of SQL statements that Oracle Database treats as a unit (it can be a single DML statement). Until a transaction is committed, it can be rolled back (undone). For more information about transactions, see "About Transaction Control Statements" on page 5-5.

Topics:



> **See Also:** *Oracle Database SQL Language Reference* for more information about DML statements

## About the INSERT Statement

The INSERT statement inserts rows into an existing table.

The simplest recommended form of the INSERT statement has this syntax:

```
INSERT INTO table_name (list_of_columns)
VALUES (list_of_values);
```

Every column in *list_of_columns* must have a valid value in the corresponding position in *list_of_values*. Therefore, before you insert a row into a table, you must know what columns the table has, and what their valid values are. To get this information using SQL Developer, see "Tutorial: Viewing EMPLOYEES Table Properties and Data" on page 3-2. To get this information using SQL*Plus, use the DESCRIBE statement. For example:

```
DESCRIBE EMPLOYEES;
```

Result:

```
 Name                                    Null?    Type
 ----------------------------------------- -------- ------------

 EMPLOYEE_ID                             NOT NULL NUMBER(6)
 FIRST_NAME                                       VARCHAR2(20)
 LAST_NAME                               NOT NULL VARCHAR2(25)
 EMAIL                                   NOT NULL VARCHAR2(25)
 PHONE_NUMBER                                     VARCHAR2(20)
 HIRE_DATE                               NOT NULL DATE
 JOB_ID                                  NOT NULL VARCHAR2(10)
 SALARY                                           NUMBER(8,2)
 COMMISSION_PCT                                   NUMBER(2,2)
 MANAGER_ID                                       NUMBER(6)
 DEPARTMENT_ID                                    NUMBER(4)
```

The INSERT statement in Example 5–1 inserts a row into the EMPLOYEES table for an employee for which all column values are known.

***Example 5–1 Using the INSERT Statement When All Information Is Available***

```
INSERT INTO EMPLOYEES (
  EMPLOYEE_ID,
  FIRST_NAME,
  LAST_NAME,
  EMAIL,
  PHONE_NUMBER,
  HIRE_DATE,
  JOB_ID,
  SALARY,
  COMMISSION_PCT,
  MANAGER_ID,
  DEPARTMENT_ID
)
VALUES (
  10,              -- EMPLOYEE_ID
  'George',        -- FIRST_NAME
  'Gordon',        -- LAST_NAME
  'GGORDON',       -- EMAIL
  '650.506.2222',  -- PHONE_NUMBER
  '01-JAN-07',     -- HIRE_DATE
  'SA_REP',        -- JOB_ID
  9000,            -- SALARY
```

```
  .1,              -- COMMISSION_PCT
  148,             -- MANAGER_ID
  80               -- DEPARTMENT_ID
);
```

Result:

```
1 row created.
```

You do not need to know all column values to insert a row into a table, but you must know the values of all NOT NULL columns. If you do not know the value of a column that can be NULL, you can omit that column from *list_of_columns*. Its value defaults to NULL.

The INSERT statement in Example 5–2 inserts a row into the EMPLOYEES table for an employee for which all column values are known except SALARY. For now, SALARY can have the value NULL. When you know the salary, you can change it with the UPDATE statement (see Example 5–4).

***Example 5–2 Using the INSERT Statement When Not All Information Is Available***

```
INSERT INTO EMPLOYEES (
  EMPLOYEE_ID,
  FIRST_NAME,
  LAST_NAME,
  EMAIL,
  PHONE_NUMBER,
  HIRE_DATE,
  JOB_ID,          -- Omit SALARY; its value defaults to NULL.
  COMMISSION_PCT,
  MANAGER_ID,
  DEPARTMENT_ID
)
VALUES (
  20,              -- EMPLOYEE_ID
  'John',          -- FIRST_NAME
  'Keats',         -- LAST_NAME
  'JKEATS',        -- EMAIL
  '650.506.3333',  -- PHONE_NUMBER
  '01-JAN-07',     -- HIRE_DATE
  'SA_REP',        -- JOB_ID
  .1,              -- COMMISSION_PCT
  148,             -- MANAGER_ID
  80               -- DEPARTMENT_ID
);
```

Result:

```
1 row created.
```

The INSERT statement in Example 5–3 tries to insert a row into the EMPLOYEES table for an employee for which LAST_NAME is not known.

***Example 5–3 Using the INSERT Statement Incorrectly***

```
INSERT INTO EMPLOYEES (
  EMPLOYEE_ID,
  FIRST_NAME,      -- Omit LAST_NAME (error)
  EMAIL,
  PHONE_NUMBER,
  HIRE_DATE,
```

```
  JOB_ID,
  COMMISSION_PCT,
  MANAGER_ID,
  DEPARTMENT_ID
)
VALUES (
  20,              -- EMPLOYEE_ID
  'John',          -- FIRST_NAME
  'JOHN',          -- EMAIL
  '650.506.3333',  -- PHONE_NUMBER
  '01-JAN-07',     -- HIRE_DATE
  'SA_REP',        -- JOB_ID
  .1,              -- COMMISSION_PCT
  148,             -- MANAGER_ID
  80               -- DEPARTMENT_ID
);
```

Result:

```
ORA-01400: cannot insert NULL into ("HR"."EMPLOYEES"."LAST_NAME")
```

**See Also:**

- *Oracle Database SQL Language Reference* for information about the `INSERT` statement
- *Oracle Database SQL Language Reference* for information about data types
- "Tutorial: Adding Rows to Tables with the Insert Row Tool" on page 6-11

## About the UPDATE Statement

The `UPDATE` statement updates (changes the values of) a set of existing table rows.

A simple form of the `UPDATE` statement has this syntax:

```
UPDATE table_name
SET column_name = value [, column_name = value]...
[ WHERE condition ];
```

Each *`value`* must be valid for its *`column_name`*. If you include the `WHERE` clause, the statement updates column values only in rows that satisfy *`condition`*.

The `UPDATE` statement in Example 5–4 updates the value of the `SALARY` column in the row that was inserted into the `EMPLOYEES` table in Example 5–2, before the salary of the employee was known.

***Example 5–4 Using the UPDATE Statement to Add Data***

```
UPDATE EMPLOYEES
SET SALARY = 8500
WHERE LAST_NAME = 'Keats';
```

Result:

```
1 row updated.
```

The `UPDATE` statement in Example 5–5 updates the commission percentage for every employee in department 80.

***Example 5–5 Using the UPDATE Statement to Update Multiple Rows***

```
UPDATE EMPLOYEES
SET COMMISSION_PCT = COMMISSION_PCT + 0.05
WHERE DEPARTMENT_ID = 80;
```

Result:

```
36 rows updated.
```

**See Also:**

- *Oracle Database SQL Language Reference* for information about the `UPDATE` statement
- *Oracle Database SQL Language Reference* for information about data types
- "Tutorial: Changing Data in Tables in the Data Pane" on page 6-12

## About the DELETE Statement

The `DELETE` statement deletes rows from a table.

A simple form of the `DELETE` statement has this syntax:

```
DELETE FROM table_name
[ WHERE condition ];
```

If you include the `WHERE` clause, the statement deletes only rows that satisfy *`condition`*. If you omit the `WHERE` clause, the statement deletes all rows from the table, but the empty table still exists. To delete a table, use the `DROP TABLE` statement.

The `DELETE` statement in Example 5–6 deletes the rows inserted in Example 5–1 and Example 5–2.

***Example 5–6 Using the DELETE Statement***

```
DELETE FROM EMPLOYEES
WHERE HIRE_DATE = '01-JAN-07';
```

Result:

```
2 rows deleted.
```

**See Also:**

- *Oracle Database SQL Language Reference* for information about the `DELETE` statement
- *Oracle Database SQL Language Reference* for information about the `DROP TABLE` statement
- "Tutorial: Deleting Rows from Tables with the Delete Selected Row(s) Tool" on page 6-13

# About Transaction Control Statements

A **transaction** is a sequence of one or more SQL statements that Oracle Database treats as a unit: either all of the statements are performed, or none of them are.

You need transactions to model business processes that require that several operations be performed as a unit. For example, when a manager leaves the company, a row must be inserted into the `JOB_HISTORY` table to show when the manager left, and for every

employee who reports to that manager, the value of MANAGER_ID must be updated in the EMPLOYEES table. To model this process in an application, you must group the INSERT and UPDATE statements into a single transaction.

The basic **transaction control statements** are:

- SAVEPOINT, which marks a **savepoint** in a transaction—a point to which you can later roll back. Savepoints are optional, and a transaction can have multiple savepoints.
- COMMIT, which ends the current transaction, makes its changes permanent, erases its savepoints, and releases its locks.
- ROLLBACK, which rolls back (undoes) either the entire current transaction or only the changes made after the specified savepoint.

In the SQL*Plus environment, you can enter a transaction control statement after the SQL> prompt.

In the SQL Developer environment, you can enter a transaction control statement in the SQL Worksheet. SQL Developer also has Commit Changes and Rollback Changes icons, which are explained in "Committing Transactions" on page 5-6 and "Rolling Back Transactions" on page 5-8.

---

**Caution:** If you do not explicitly commit a transaction, and the program terminates abnormally, then the database automatically rolls back the last uncommitted transaction.

Oracle recommends that you explicitly end transactions in application programs, by either committing them or rolling them back.

---

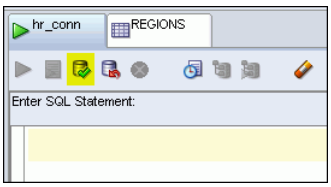
**See Also:**

- *Oracle Database Concepts* for more information about transaction management
- *Oracle Database SQL Language Reference* for more information about transaction control statements

## Committing Transactions

Committing a transaction makes its changes permanent, erases its savepoints, and releases its locks.

To explicitly commit a transaction, use either the COMMIT statement or (in the SQL Developer environment) the Commit Changes icon.




---

**Note:** Oracle Database issues an implicit COMMIT statement before and after any data definition language (DDL) statement. For information about DDL statements, see "About Data Definition Language (DDL) Statements" on page 6-1.

---

Before you commit a transaction:

- Your changes are visible to you, but not to other users of the database instance.
- Your changes are not final—you can undo them with a `ROLLBACK` statement.

After you commit a transaction:

- Your changes are visible to other users, and to their statements that run after you commit your transaction.
- Your changes are final—you cannot undo them with a `ROLLBACK` statement.

Example 5–7 adds one row to the `REGIONS` table (a very simple transaction), checks the result, and then commits the transaction.

***Example 5–7 Committing a Transaction***

Before transaction:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa
```

Transaction (add row to table):

```
INSERT INTO regions (region_id, region_name) VALUES (5, 'Africa');
```

Result:

```
1 row created.
```

Check that row was added:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa
         5 Africa
```

Commit transaction:

```
COMMIT;
```

Result:

```
Commit complete.
```

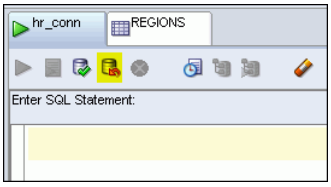
> **See Also:** *Oracle Database SQL Language Reference* for information about the `COMMIT` statement

# Rolling Back Transactions

Rolling back a transaction undoes its changes. You can roll back the entire current transaction, or you can roll it back only to a specified savepoint.

To roll back the current transaction only to a specified savepoint, you must use the `ROLLBACK` statement with the `TO SAVEPOINT` clause.

To roll back the entire current transaction, use either the `ROLLBACK` statement without the `TO SAVEPOINT` clause, or (in the SQL Developer environment) the Rollback Changes icon.




Rolling back the entire current transaction:

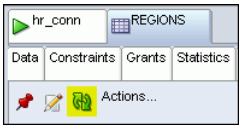
- Ends the transaction
- Reverses all of its changes
- Erases all of its savepoints
- Releases any transaction locks

Rolling back the current transaction only to the specified savepoint:

- Does not end the transaction
- Reverses only the changes made after the specified savepoint
- Erases only the savepoints set after the specified savepoint (excluding the specified savepoint itself)
- Releases all table and row locks acquired after the specified savepoint

  Other transactions that have requested access to rows locked after the specified savepoint must continue to wait until the transaction is either committed or rolled back. Other transactions that have not requested the rows can request and access the rows immediately.

To see the effect of a rollback in SQL Developer, you might have to click the Refresh icon.




As a result of Example 5–7, the `REGIONS` table has a region called 'Middle East and Africa' and a region called 'Africa'. Example 5–8 corrects this problem (a very simple transaction) and checks the change, but then rolls back the transaction and checks the rollback.

***Example 5–8 Rolling Back an Entire Transaction***

Before transaction:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa
         5 Africa
```

Transaction (change table):

```
UPDATE REGIONS
SET REGION_NAME = 'Middle East'
WHERE REGION_NAME = 'Middle East and Africa';
```

Result:

```
1 row updated.
```

Check change:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East
         5 Africa
```

Roll back transaction:

```
ROLLBACK;
```

Result:

```
Rollback complete.
```

Check rollback:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa
         5 Africa
```

**See Also:** *Oracle Database SQL Language Reference* for information about the `ROLLBACK` statement

## Setting Savepoints in Transactions

The SAVEPOINT statement marks a **savepoint** in a transaction—a point to which you can later roll back. Savepoints are optional, and a transaction can have multiple savepoints.

Example 5–9 does a transaction that includes several DML statements and several savepoints, and then rolls back the transaction to one savepoint, undoing only the changes made after that savepoint.

***Example 5–9 Rolling Back a Transaction to a Savepoint***

Check REGIONS table before transaction:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East and Africa
         5 Africa

5 rows selected.
```

Check countries in region 4 before transaction:

```
SELECT COUNTRY_NAME, COUNTRY_ID, REGION_ID
FROM COUNTRIES
WHERE REGION_ID = 4
ORDER BY COUNTRY_NAME;
```

Result:

```
COUNTRY_NAME                             CO  REGION_ID
---------------------------------------- -- ----------
Egypt                                    EG          4
Israel                                   IL          4
Kuwait                                   KW          4
Nigeria                                  NG          4
Zambia                                   ZM          4
Zimbabwe                                 ZW          4

6 rows selected.
```

Check countries in region 5 before transaction:

```
SELECT COUNTRY_NAME, COUNTRY_ID, REGION_ID
FROM COUNTRIES
WHERE REGION_ID = 5
ORDER BY COUNTRY_NAME;
```

Result:

```
no rows selected
```

Transaction, with several savepoints:

```
UPDATE REGIONS
```

```
SET REGION_NAME = 'Middle East'
WHERE REGION_NAME = 'Middle East and Africa';

UPDATE COUNTRIES
  SET REGION_ID = 5
  WHERE COUNTRY_ID = 'ZM';
SAVEPOINT zambia;

UPDATE COUNTRIES
  SET REGION_ID = 5
  WHERE COUNTRY_ID = 'NG';
SAVEPOINT nigeria;

UPDATE COUNTRIES
  SET REGION_ID = 5
  WHERE COUNTRY_ID = 'ZW';
SAVEPOINT zimbabwe;

UPDATE COUNTRIES
  SET REGION_ID = 5
  WHERE COUNTRY_ID = 'EG';
SAVEPOINT egypt;
```

Check `REGIONS` table after transaction:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East
         5 Africa

5 rows selected.
```

Check countries in region 4 after transaction:

```
SELECT COUNTRY_NAME, COUNTRY_ID, REGION_ID
FROM COUNTRIES
WHERE REGION_ID = 4
ORDER BY COUNTRY_NAME;
```

Result:

```
COUNTRY_NAME                             CO  REGION_ID
---------------------------------------- -- ----------
Israel                                   IL          4
Kuwait                                   KW          4

2 rows selected.
```

Check countries in region 5 after transaction:

```
SELECT COUNTRY_NAME, COUNTRY_ID, REGION_ID
FROM COUNTRIES
WHERE REGION_ID = 5
ORDER BY COUNTRY_NAME;
```

Result:

```
COUNTRY_NAME                             CO  REGION_ID
---------------------------------------- -- ----------
Egypt                                    EG          5
Nigeria                                  NG          5
Zambia                                   ZM          5
Zimbabwe                                 ZW          5

4 rows selected.
```

```
ROLLBACK TO SAVEPOINT nigeria;
```

Check `REGIONS` table after rollback:

```
SELECT * FROM REGIONS
ORDER BY REGION_ID;
```

Result:

```
 REGION_ID REGION_NAME
---------- -------------------------
         1 Europe
         2 Americas
         3 Asia
         4 Middle East
         5 Africa

5 rows selected.
```

Check countries in region 4 after rollback:

```
SELECT COUNTRY_NAME, COUNTRY_ID, REGION_ID
FROM COUNTRIES
WHERE REGION_ID = 4
ORDER BY COUNTRY_NAME;
```

Result:

```
COUNTRY_NAME                             CO  REGION_ID
---------------------------------------- -- ----------
Egypt                                    EG          4
Israel                                   IL          4
Kuwait                                   KW          4
Zimbabwe                                 ZW          4

4 rows selected.
```

Check countries in region 5 after rollback:

```
SELECT COUNTRY_NAME, COUNTRY_ID, REGION_ID
FROM COUNTRIES
WHERE REGION_ID = 5
ORDER BY COUNTRY_NAME;
```

Result:

```
COUNTRY_NAME                             CO  REGION_ID
---------------------------------------- -- ----------
Nigeria                                  NG          5
Zambia                                   ZM          5
```

```
2 rows selected.
```

**See Also:** *Oracle Database SQL Language Reference* for information about the `SAVEPOINT` statement

# 6

# Creating and Managing Schema Objects

This chapter contains the following topics:



## About Data Definition Language (DDL) Statements

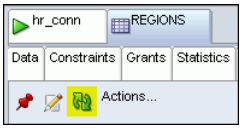
The statements that create, change, and drop schema objects are **data definition language (DDL) statements**. Before and after a DDL statement, Oracle Database issues an implicit `COMMIT` statement; therefore, you cannot roll back a DDL statement.

In the SQL*Plus environment, you can enter a DDL statement after the `SQL>` prompt.

In the SQL Developer environment, you can enter a DDL statement in the SQL Worksheet. Alternatively, you can use SQL Developer tools to create, change, and drop objects.

Some DDL statements that create schema objects have an optional `OR REPLACE` clause, which allows a statement to replace an existing schema object with another that has the same name and type. When SQL Developer generates code for one of these statements, it always includes the `OR REPLACE` clause.

To see the effect of a DDL statement in SQL Developer, you might have to click the Refresh icon.




**See Also:**

- *Oracle Database SQL Language Reference* for more information about DDL statements
- "Committing Transactions" on page 5-6

# About Schema Object Names

When creating schema objects, you must observe the schema object naming rules in *Oracle Database SQL Language Reference*.

> **Tip:** Use the same prefix for names of objects of the same type. For example, `t_` for tables, `v_` for views, `seq_` for sequences, and `syn_` for synonyms. This practice makes your objects easy to identify, and groups them in the SQL Developer Connections navigator display, SQL Developer reports, and queries whose results are ordered by object name.

# Creating and Managing Tables

Tables are the basic units of data storage in Oracle Database. Tables hold all user-accessible data. Each table contains rows that represent individual data records. Rows are composed of columns that represent the fields of the records.

Topics:

- About SQL Data Types
- Creating Tables
- Ensuring Data Integrity in Tables
- Tutorial: Adding Rows to Tables with the Insert Row Tool
- Tutorial: Changing Data in Tables in the Data Pane
- Tutorial: Deleting Rows from Tables with the Delete Selected Row(s) Tool
- Managing Indexes
- Dropping Tables

---

**Note:** To do the tutorials in this document, you must be connected to Oracle Database as the user `HR` from SQL Developer. For instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6.

---

**See Also:**

- "Tutorial: Viewing EMPLOYEES Table Properties and Data" on page 3-2
- *Oracle Database Concepts* for general information about tables

## About SQL Data Types

When you create a table, you must specify the SQL data type for each column. The data type of a column determines what values the column can contain. For example, a column of type `DATE` can contain the value `'01-MAY-05'`, but it cannot contain the numeric value 2 or the character value `'shoe'`. SQL data types fall into two categories: built-in and user-defined. (PL/SQL has additional data types—see "About PL/SQL Data Types" on page 7-4.)

**See Also:**

- *Oracle Database SQL Language Reference* for a summary of built-in SQL data types
- *Oracle Database Concepts* for introductions to each of the built-in SQL data types
- *Oracle Database SQL Language Reference* for more information about user-defined data types
- "About PL/SQL Data Types" on page 7-4

## Creating Tables

To create tables, use either the SQL Developer tool Create Table or the DDL statement `CREATE TABLE`. This topic shows how to use both of these ways to create these tables, which will contain data about employee evaluations:

- `PERFORMANCE_PARTS`, which contains the categories of employee performance that are evaluated and their relative weights
- `EVALUATIONS`, which contains employee information, evaluation date, job, manager, and department
- `SCORES`, which contains the scores assigned to each performance category for each evaluation

These tables are part of the sample application that the tutorials and examples in this document show how to develop and deploy.

Topics:



### Tutorial: Creating a Table with the Create Table Tool

This tutorial shows how to create the `PERFORMANCE_PARTS` table using the Create Table tool.

#### To create the PERFORMANCE_PARTS table using the Create Table tool:

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Right-click **Tables**.

   A list of choices appears.

3. Click **New Table**.

   The Create Table window opens, with default values for a new table, which has only one row.

4. For Schema, accept the default value, `HR`.
5. For Name, enter `PERFORMANCE_PARTS`.
6. In the default row:
   - For Column Name, enter `PERFORMANCE_ID`.
   - For Type, accept the default value, `VARCHAR2`.

- For Size, enter 2.
- For Not Null and Primary Key, accept the default values, deselected.

7. Click **Add Column**.
8. For Column Name, enter NAME.
9. For Type, enter VARCHAR2.
10. For Size, enter 80.
11. Click **Add Column**.
12. For Column Name, enter WEIGHT.
13. For Type, enter NUMBER.
14. Click **OK**.

    The table PERFORMANCE_PARTS is created. To see it, expand Tables in the navigation frame.

    **See Also:** *Oracle Database SQL Developer User's Guide* for more information about using SQL Developer to create tables

### Creating Tables with the CREATE TABLE Statement

This topic shows how to use the CREATE TABLE statement to create the EVALUATIONS and SCORES tables.

The CREATE TABLE statement in Example 6–1 creates the EVALUATIONS table.

***Example 6–1 Creating the EVALUATIONS Table with CREATE TABLE***

```
CREATE TABLE EVALUATIONS (
  EVALUATION_ID    NUMBER(8,0),
  EMPLOYEE_ID      NUMBER(6,0),
  EVALUATION_DATE  DATE,
  JOB_ID           VARCHAR2(10),
  MANAGER_ID       NUMBER(6,0),
  DEPARTMENT_ID    NUMBER(4,0),
  TOTAL_SCORE      NUMBER(3,0)
);
```

Result:

```
Table created.
```

The CREATE TABLE statement in Example 6–2 creates the SCORES table.

***Example 6–2 Creating the SCORES Table with CREATE TABLE***

```
CREATE TABLE SCORES (
  EVALUATION_ID   NUMBER(8,0),
  PERFORMANCE_ID  VARCHAR2(2),
  SCORE           NUMBER(1,0)
);
```

Result:

```
Table created.
```

In SQL Developer, in the navigation frame, if you expand Tables, you can see the tables EVALUATIONS and SCORES.

If you select a table in the navigation frame, and then click the tab SQL in the right frame, the SQL pane shows the SQL statement that created the table.

> **See Also:** *Oracle Database SQL Language Reference* for information about the `CREATE TABLE` statement

## Ensuring Data Integrity in Tables

To ensure that the data in your tables satisfies the business rules that your application models, you can use constraints, application logic, or both.

**Constraints** restrict the values that columns can have. Trying to change the data in a way that violates a constraint causes an error and rolls back the change. Trying to add a constraint to a populated table causes an error if existing data violates the constraint.

> **Tip:** Wherever possible, use constraints instead of application logic. Oracle Database checks that all data obeys constraints much faster than application logic can.

Constraints can be enabled and disabled. By default, they are created in the enabled state.

Topics:

- About Constraint Types
- Tutorial: Adding Constraints to Existing Tables

> **See Also:**
>
> - *Oracle Database Concepts* for additional general information about constraints
> - *Oracle Database SQL Language Reference* for syntactic information about constraints
> - "Installing the Sample Application" on page 10-12

### About Constraint Types

The constraint types are:

- **Not Null**, which prevents a value from being null

  In the `EMPLOYEES` table, the column `LAST_NAME` has the `NOT NULL` constraint, which enforces the business rule that every employee must have a last name.

- **Unique**, which prevents multiple rows from having the same value in the same column or combination of columns, but allows some values to be null

  In the `EMPLOYEES` table, the column `EMAIL` has the `UNIQUE` constraint, which enforces the business rule that an employee can have no email address, but cannot have the same email address as another employee.

- **Primary Key**, which is a combination of `NOT NULL` and `UNIQUE`

  In the `EMPLOYEES` table, the column `EMPLOYEE_ID` has the `PRIMARY KEY` constraint, which enforces the business rule that every employee must have a unique employee identification number.

- **Foreign Key**, which requires values in one table to match values in another table

In the EMPLOYEES table, the column JOB_ID has a FOREIGN KEY constraint that references the JOBS table, which enforces the business rule that an employee cannot have a JOB_ID that is not in the JOBS table.

- **Check**, which requires that a value satisfy a specified condition

  The EMPLOYEES table does not have CHECK constraints. However, suppose that EMPLOYEES needs a new column, EMPLOYEE_AGE, and that every employee must be at least 18. The constraint CHECK (EMPLOYEE_AGE >= 18) enforces the business rule.

  **Tip:** Use check constraints only when other constraint types cannot provide the necessary checking.

- **REF**, which further describes the relationship between the column and the object that it references

  For information about REF constraints, see *Oracle Database Concepts*.

  **See Also:**

  - *Oracle Database Concepts* for additional general information about constraint types
  - *Oracle Database SQL Language Reference* for syntactic information about constraints

### Tutorial: Adding Constraints to Existing Tables

To add constraints to existing tables, use either SQL Developer tools or the DDL statement ALTER TABLE. This topic shows how to use both of these ways to add constraints to the tables created in "Creating Tables" on page 6-3.

This tutorial has several procedures. The first procedure (immediately after this paragraph) uses the Edit Table tool to add a Not Null constraint to the NAMES column of the PERFORMANCE_PARTS table. The remaining procedures show how to use other tools to add constraints; however, you could add the same constraints using the Edit Table tool.

---

**Note:** After any step of the tutorial, you can view the constraints that a table has:

1. In the navigation frame, select the name of the table.
2. In the right frame, click the tab **Constraints**.

For more information about viewing table properties and data, see "Tutorial: Viewing EMPLOYEES Table Properties and Data" on page 3-2.

---

#### To add a Not Null constraint using the Edit Table tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears.

3. Right-click **PERFORMANCE_PARTS**.

   A list of choices appears.

4. Click **Edit**.

   The Edit Table window opens. By default, Columns is selected, the columns of the PERFORMANCE_PARTS table are listed, the column PERFORMANCE_ID is selected, and its properties are listed.

5. Click the column **NAME**.

   The properties of the column NAME appear. The property "Cannot be NULL" is deselected.

6. Select **Cannot be NULL**.

7. Click **OK**.

   The Not Null constraint is added to the NAME column of the PERFORMANCE_PARTS table.

The following procedure uses the ALTER TABLE statement to add a Not Null constraint to the WEIGHT column of the PERFORMANCE_PARTS table.

#### To add a Not Null constraint using the ALTER TABLE statement:

1. Click the icon **SQL Worksheet**.

   The SQL Worksheet pane appears.

2. In the SQL Worksheet pane, type this statement:

```
ALTER TABLE PERFORMANCE_PARTS
MODIFY WEIGHT NOT NULL;
```

3. Click the icon **Execute Statement**.

   The statement runs, adding the Not Null constraint to the WEIGHT column of the PERFORMANCE_PARTS table.

The following procedure uses the Add Unique tool to add a Unique constraint to the SCORES table.

#### To add a Unique constraint using the Add Unique tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears.

3. Right-click **SCORES**.

   A list of choices appears.

4. Click **Constraint**.

   A list of choices appears.

5. Click **Add Unique**.

   The Add Unique window opens.

6. For Constraint Name, enter SCORES_EVAL_PERF_UNIQUE.

7. For Column 1, select EVALUATION_ID from the drop-down menu.

8. For Column 2, select PERFORMANCE_ID from the drop-down menu.

9. Click **Apply**.

The Confirmation window opens.

10. Click **OK**.

    A unique constraint named SCORES_EVAL_PERF_UNIQUE is added to the SCORES table.

The following procedure uses the Add Primary Key tool to add a Primary Key constraint to the PERFORMANCE_ID column of the PERFORMANCE_PARTS table.

#### To add a Primary Key constraint using the Add Primary Key tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears.

3. Right-click **PERFORMANCE_PARTS**.

   A list of choices appears.

4. Click **Constraint**.

   A list of choices appears.

5. Click **Add Primary Key**.

   The Add Primary Key window opens.

6. For Primary Key Name, enter PERF_PERF_ID_PK.

7. For Column 1, select PERFORMANCE_ID from the drop-down menu.

8. Click **Apply**.

   The Confirmation window opens.

9. Click **OK**.

   A primary key constraint named PERF_PERF_ID_PK is added to the PERFORMANCE_ID column of the PERFORMANCE_PARTS table.

The following procedure uses the ALTER TABLE statement to add a Primary Key constraint to the EVALUATION_ID column of the EVALUATIONS table.

#### To add a Primary Key constraint using the ALTER TABLE statement:

1. Click the icon **SQL Worksheet**.

   The SQL Worksheet pane appears. Under "Enter SQL Statement:" is a field where you can enter a SQL statement.

2. In the SQL Worksheet pane, type this statement:

```
ALTER TABLE EVALUATIONS
ADD CONSTRAINT EVAL_EVAL_ID_PK PRIMARY KEY (EVALUATION_ID);
```

3. Click the icon **Execute Statement**.

   The statement runs, adding the Primary Key constraint to the EVALUATION_ID column of the EVALUATIONS table.

The following procedure uses the Add Foreign Key tool to add two Foreign Key constraints to the SCORES table.

**To add two Foreign Key constraints using the Add Foreign Key tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears.

3. Right-click **SCORES**.

   A list of choices appears.

4. Click **Constraint**.

   A list of choices appears.

5. Click **Add Foreign Key**.

   The Add Foreign Key window opens.

6. For Foreign Key Name, enter `SCORES_EVAL_FK`.

7. For Column Name, select `EVALUATION_ID` from the drop-down menu.

8. For Reference Table Name, select `EVALUATIONS` from the drop-down menu.

9. For Referencing Column, select `EVALUATION_ID` from the drop-down menu.

10. Click **Apply**.

    The Confirmation window opens.

11. Click **OK**.

    A foreign key constraint named `SCORES_EVAL_FK` is added to the `EVALUTION_ID` column of the `SCORES` table, referencing the `EVALUTION_ID` column of the `EVALUATIONS` table.

    The following steps add another foreign key constraint to the `SCORES` table.

12. Right-click **SCORES**.

    A list of tables appears.

13. Click **Constraint**.

    A list of choices appears.

14. Click **Add Foreign Key**.

    The Add Foreign Key window opens.

15. For Foreign Key Name, enter `SCORES_PERF_FK`.

16. For Column Name, select `PERFORMANCE_ID` from the drop-down menu.

17. For Reference Table Name, select `PERFORMANCE_PARTS` from the drop-down menu.

18. For Referencing Column, select `PERFORMANCE_ID` from the drop-down menu.

19. Click **Apply**.

    The Confirmation window opens.

20. Click **OK**.

    A foreign key constraint named `SCORES_PERF_FK` is added to the `EVALUTION_ID` column of the `SCORES` table, referencing the `EVALUTION_ID` column of the `EVALUATIONS` table.

The following procedure uses the ALTER TABLE statement to add a Foreign Key constraint to the EMPLOYEE_ID column of the EVALUATIONS table, referencing the EMPLOYEE_ID column of the EMPLOYEES table.

**To add a Foreign Key constraint using the ALTER TABLE statement:**

1. Click the icon **SQL Worksheet**.

   The SQL Worksheet pane appears. Under "Enter SQL Statement:" is a field where you can enter a SQL statement.

2. In the SQL Worksheet pane, type this statement:

```
ALTER TABLE EVALUATIONS
ADD CONSTRAINT EVAL_EMP_ID_FK FOREIGN KEY (EMPLOYEE_ID)
REFERENCES EMPLOYEES (EMPLOYEE_ID);
```

3. Click the icon **Execute Statement**.

   The statement runs, adding the Foreign Key constraint to the EMPLOYEE_ID column of the EVALUATIONS table, referencing the EMPLOYEE_ID column of the EMPLOYEES table.

The following procedure uses the Add Check tool to add a Check constraint to the SCORES table.

**To add a Check constraint using the Add Check tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears.

3. Right-click **SCORES**.

   A list of choices appears.

4. Click **Constraint**.

   A list of choices appears.

5. Click **Add Check**.

   The Add Check window opens.

6. For Constraint Name, enter SCORE_VALID.

7. For Check Condition, enter score >= 0 and score <= 9.

8. For Status, accept the default, ENABLE.

9. Click **Apply**.

   The Confirmation window opens.

10. Click **OK**.

    A Check constraint named SCORE_VALID is added to the SCORES table.

**See Also:**

- *Oracle Database SQL Language Reference* for more information about the `ALTER TABLE` statement
- *Oracle Database SQL Developer User's Guide* for information about adding constraints to a table when you create it with SQL Developer
- *Oracle Database SQL Language Reference* for information about adding constraints to a table when you create it with the `CREATE TABLE` statement

## Tutorial: Adding Rows to Tables with the Insert Row Tool

This tutorial shows how to use the Insert Row tool to add six populated rows to the `PERFORMANCE_PARTS` table (created in "Tutorial: Creating a Table with the Create Table Tool" on page 6-3).

**To add rows to the PERFORMANCE_PARTS table using the Insert Row tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears.

3. Select **PERFORMANCE_PARTS**.

4. In the right frame, click the tab **Data**.

   The Data pane appears, showing the names of the columns of the `PERFORMANCE_PARTS` table and no rows.

5. In the Data pane, click the icon **Insert Row**.

   A new row appears, with empty columns.

6. Click the cell under the column heading `PERFORMANCE_ID`.

7. Type `WM`.

   The value of `PERFORMANCE_ID` is now `WM`.

8. Either press the key **Tab** or click the cell under the column heading `NAME`.

9. Type `Workload Management`.

   The value of `NAME` is now `Workload Management`.

10. Either press the key **Tab** or click the cell under the column heading `WEIGHT`.

11. Type `0.2`.

12. Press the key **Enter**.

    The value of `WEIGHT` is now `0.2`.

13. Add and populate a second row by repeating steps 5 through 12 with these values:

    - For `PERFORMANCE_ID`, type `BR`.
    - For `NAME`, type `Building Relationships`.
    - For `WEIGHT`, type `0.2`.

14. Add and populate a third row by repeating steps 5 through 12 with these values:

- For PERFORMANCE_ID, type CF.
- For NAME, type Customer Focus.
- For WEIGHT, type 0.2.

15. Add and populate a fourth row by repeating steps 5 through 12 with these values:
    - For PERFORMANCE_ID, type CM.
    - For NAME, type Communication.
    - For WEIGHT, type 0.2.
16. Add and populate a fifth row by repeating steps 5 through 12 with these values:
    - For PERFORMANCE_ID, type TW.
    - For NAME, type Teamwork.
    - For WEIGHT, type 0.2.
17. Add and populate a sixth row by repeating steps 5 through 12, using these values:
    - For PERFORMANCE_ID, type RO.
    - For NAME, type Results Orientation.
    - For WEIGHT, type 0.2.
18. Click the icon **Commit Changes**.

    Under the Data pane is the Data Editor Log pane.
19. Check the Data Editor Log pane for the message "Commit Successful".
20. Check the new rows in the Data Pane.

**See Also:** "About the INSERT Statement" on page 5-2

## Tutorial: Changing Data in Tables in the Data Pane

This tutorial shows how to change three of the WEIGHT values in the PERFORMANCE_PARTS table (populated in "Tutorial: Adding Rows to Tables with the Insert Row Tool" on page 6-11) in the Data pane.

### To change data in the PERFORMANCE_PARTS table using the Data pane:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.
2. Expand **Tables**.

   A list of tables appears.
3. Select **PERFORMANCE_PARTS**.
4. In the right frame, click the tab **Data**.

   The Data pane appears, showing the rows of the PERFORMANCE_PARTS table.
5. In the row where NAME is Workload Management:
   1. Click the WEIGHT value.
   2. Enter the value 0.3.
   3. Press the key **Enter**.
6. In the row where NAME is Building Relationships:

1. Click the WEIGHT value.
2. Enter the value 0.15.
3. Press the key **Enter**.

7. In the row where NAME is Customer Focus:
   1. Click the WEIGHT value.
   2. Enter the value 0.15.
   3. Press the key **Enter**.
8. Click the icon **Commit Changes**.

   This icon is a picture of a data drum with a green check mark in the lower right corner.

   Under the Data pane is the Data Editor Log pane.
9. Check the Data Editor Log pane for the message "Commit Successful".
10. Check the new data in the Data Pane.

**See Also:** "About the UPDATE Statement" on page 5-4

## Tutorial: Deleting Rows from Tables with the Delete Selected Row(s) Tool

This tutorial shows how to use the Delete Selected Row(s) tool to delete a row from the PERFORMANCE_PARTS table (populated in "Tutorial: Adding Rows to Tables with the Insert Row Tool" on page 6-11).

### To delete row from PERFORMANCE_PARTS using Delete Selected Row(s) tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.
2. Expand **Tables**.

   A list of tables appears.
3. Select **PERFORMANCE_PARTS**.
4. In the right frame, click the tab **Data**.

   The Data pane appears, showing the rows of the PERFORMANCE_PARTS table.
5. Click the row where NAME is Results Orientation.
6. Click the icon **Delete Selected Row(s)**.
7. Click the icon **Commit Changes**.

   The row is deleted.

   Under the Data pane is the Data Editor - Log pane.
8. Check the Data Editor Log pane for the message "Commit Successful".

---

**Note:** If you delete every row of a table, the empty table still exists. To delete a table, see "Dropping Tables" on page 6-16.

---

**See Also:** "About the DELETE Statement" on page 5-5

## Managing Indexes

You can create indexes on one or more columns of a table to speed SQL statement execution on that table. When properly used, indexes are the primary means of reducing disk I/O.

When you define a primary key on a table, Oracle Database creates a Unique index on the primary key. For example, in "Tutorial: Adding Constraints to Existing Tables" on page 6-6, you added a Primary Key constraint to the `EVALUATION_ID` column of the `EVALUATIONS` table. Therefore, if you select the `EVALUATIONS` table in the SQL Developer navigation frame and click the Indexes tab, the Indexes pane shows a Unique index on the `EVALUATION_ID` column.

Topics:

- Tutorial: Adding an Index with the Create Index Tool
- Tutorial: Changing an Index with the Edit Index Tool
- Tutorial: Dropping an Index

**See Also:** For more information about indexes:

- *Oracle Database Concepts*
- *Oracle Database Advanced Application Developer's Guide*

### Tutorial: Adding an Index with the Create Index Tool

To create an index, use either the SQL Developer tool Create Index or the DDL statement `CREATE INDEX`.

This tutorial shows how to use the Create Index tool to add an index to the `EVALUATIONS` table (created in "Creating Tables with the CREATE TABLE Statement" on page 6-4). The equivalent DDL statement is:

```
CREATE INDEX EVAL_JOB_IX
ON EVALUATIONS (JOB_ID ASC) NOPARALLEL;
```

#### To add an index to the EVALUATIONS table using the Create Index tool:

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears, including `EVALUATIONS`.

3. Right-click **EVALUATIONS**.

   A list of choices appears.

4. Select **Index**.

   A list of choices appears.

5. Select **Create Index**.

   The Create Index window opens.

6. For Schema, accept the default, `HR`.

7. For Name, type `EVAL_JOB_IX`.

8. Select the tab **Definition**.

   The Definition pane shows the default values for index properties.

9. In the field labeled "Column Name or Expression:", type JOB_ID.

   (For all other properties, accept the default values.)

10. Click **OK**.

    Now the EVALUATIONS table has an index named EVAL_JOB_IX on the column JOB_ID.

> **See Also:** *Oracle Database SQL Language Reference* for information about the CREATE INDEX statement

### Tutorial: Changing an Index with the Edit Index Tool

To change an index, use either the SQL Developer tool Edit Index or the DDL statements DROP INDEX and CREATE INDEX.

This tutorial shows how to use the Edit Index tool to reverse the sort order of the index EVAL_JOB_IX (created in "Tutorial: Adding an Index with the Create Index Tool" on page 6-14). The equivalent DDL statements are:

```
DROP INDEX EVAL_JOB_ID;

CREATE INDEX EVAL_JOB_IX
ON EVALUATIONS (JOB_ID DESC) NOPARALLEL;
```

#### To reverse the sort order of the index EVAL_JOB_IX using the Edit Index tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Indexes**.

   A list of indexes appears, including EVAL_JOB_IX.

3. Right-click **EVAL_JOB_IX**.

   A list of choices appears.

4. Click **Edit**.

   A list of choices appears.

5. Click **Edit Index**.

   The Edit Index window opens.

6. In the Edit Index window, change **Order** to DESC.

7. Click **OK**.

> **See Also:** *Oracle Database SQL Language Reference* for information about the ALTER INDEX statement

### Tutorial: Dropping an Index

To drop an index, use either the SQL Developer tool Drop or the DDL statement DROP INDEX.

This tutorial shows how to use the navigation frame and Drop tool to drop the index EVAL_JOB_IX (created in "Tutorial: Adding an Index with the Create Index Tool" on page 6-14). The equivalent DDL statement is:

```
DROP INDEX EVAL_JOB_ID;
```

**To drop the index EVAL_JOB_IX:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Indexes**.

   A list of indexes appears, including `EVAL_JOB_IX`.

3. Right-click **EVAL_JOB_IX**.

   A list of choices appears.

4. Click **Drop**.

   The Drop window opens.

5. Click **Apply**.

   The Confirmation window opens.

6. Click **OK**.

**See Also:** *Oracle Database SQL Language Reference* for information about the `DROP INDEX` statement

## Dropping Tables

To drop a table, use either the SQL Developer navigation frame and Drop tool, or the DDL statement `DROP TABLE`.

**Caution:** Do not drop any of the tables that you created in "Creating Tables" on page 6-3—you need them for later tutorials. If you want to practice dropping tables, create simple ones and then drop them.

**To drop a table using the Drop tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Tables**.

   A list of tables appears.

3. Right-click the name of the table to drop.

   A list of choices appears.

4. Click **Table**.

   A list of choices appears.

5. Click **Drop**.

   The Drop window opens.

6. Click **Apply**.

   The Confirmation window opens.

7. Click **OK**.

**See Also:** *Oracle Database SQL Language Reference* for information about the `DROP TABLE` statement

# Creating and Managing Views

A view presents the output of a query as a table. In most places that you can use a table, you can use a view. Views are useful when you need frequent access to information that is stored in several different tables.

Topics:

- Creating Views
- Tutorial: Changing Views with the Edit View and Rename Tools
- Dropping Views

**See Also:**

- Chapter 4, "Selecting Table Data," for information about queries
- *Oracle Database Concepts* for additional general information about views

## Creating Views

To create views, use either the SQL Developer tool Create View or the DDL statement `CREATE VIEW`. This topic shows how to use both of these ways to create these views:

- `SALESFORCE`, which contains the names and salaries of the employees in the Sales department
- `EMP_LOCATIONS`, which contains the names and locations of all employees

These view are part of the sample application that the tutorials and examples in this document show how to develop and deploy.

Topics:

- Tutorial: Creating a View with the Create View Tool
- Creating Views with the CREATE VIEW Statement

**See Also:**

- *Oracle Database SQL Developer User's Guide* for more information about using SQL Developer to create a view
- *Oracle Database SQL Language Reference* for more information about the `CREATE VIEW` statement

### Tutorial: Creating a View with the Create View Tool

This tutorial shows how to create the `SALESFORCE` view using the Create View tool.

**To create the SALESFORCE view using the Create View tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Right-click **Views**.

   A list of choices appears.

3. Click **New View**.

   The Create View window opens, with default values for a new view.

4. For Schema, accept the default value, `HR`.

5. For Name, enter SALESFORCE.
6. Click the tab **SQL Query**.

   The SQL Query pane appears.
7. In the SQL Query field:
   - After SELECT, type:

     ```
     FIRST_NAME || ' ' || LAST_NAME "Name", SALARY*12 "Annual Salary"
     ```
   - After FROM, type:

     ```
     EMPLOYEES WHERE DEPARTMENT_ID = 80
     ```
8. Click **Test Syntax**.

   A message appears in the field SQL Parse Results.
9. If the message is not "No errors found in SQL", return to step 7 and correct the syntax errors in the query.
10. Click **OK**.

    The view SALESFORCE is created. To see it, expand Views in the navigation frame.

> **See Also:** *Oracle Database SQL Developer User's Guide* for more information about using SQL Developer to create views

### Creating Views with the CREATE VIEW Statement

The CREATE VIEW statement in Example 6–3 creates the EMP_LOCATIONS view, which joins four tables. (For information about joins, see Section , "Selecting Data from Multiple Tables.")

***Example 6–3 Creating the EMP_LOCATIONS View with CREATE VIEW***

```
CREATE VIEW EMP_LOCATIONS AS
SELECT e.EMPLOYEE_ID,
  e.LAST_NAME || ', ' || e.FIRST_NAME NAME,
  d.DEPARTMENT_NAME DEPARTMENT,
  l.CITY CITY,
  c.COUNTRY_NAME COUNTRY
FROM EMPLOYEES e, DEPARTMENTS d, LOCATIONS l, COUNTRIES c
WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID AND
 d.LOCATION_ID = l.LOCATION_ID AND
 l.COUNTRY_ID = c.COUNTRY_ID
ORDER BY LAST_NAME;
```

Result:

```
View created.
```

> **See Also:** *Oracle Database SQL Language Reference* for information about the CREATE VIEW statement

## Tutorial: Changing Views with the Edit View and Rename Tools

To change the query in a view, use either the SQL Developer tool Edit View or the DDL statement CREATE VIEW with the OR REPLACE clause. To change the name of a view, use either the Rename tool or the RENAME statement.

This tutorial shows how to use the Edit View tool to add the employees of the Marketing department to the SALESFORCE view (created in "Tutorial: Creating a View with the Create View Tool" on page 6-17) and the and Rename tool to change its name to SALES_MARKETING. The equivalent DDL statements are:

```
CREATE OR REPLACE VIEW SALESFORCE AS
  SELECT FIRST_NAME || ' ' || LAST_NAME "Name",
  SALARY*12 "Annual Salary"
  FROM EMPLOYEES
  WHERE DEPARTMENT_ID = 80 OR DEPARTMENT_ID = 20;

RENAME SALESFORCE to SALES_MARKETING;
```

### To change the SALESFORCE view using the Edit View and Rename tools:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Views**.

   A list of views appears.

3. Right-click **SALESFORCE**.

   A list of choices appears.

4. Select **Edit**.

   The Edit View window opens. Its SQL Query field contains the query to be changed.

5. Add this to the end of the query:

   ```
   OR DEPARTMENT_ID = 20
   ```

6. Click **Test Syntax**.

   A message appears in the field SQL Parse Results.

7. If the message is not "No errors found in SQL", return to step 5 and correct the syntax errors in the query.

8. Click **OK**.

   The employees of the Marketing department are added to the SALESFORCE view. Now change the name of the view to SALES_MARKETING.

9. Right-click **SALESFORCE**.

   A list of choices appears.

10. Select **Rename**.

    The Rename window opens. It has a New View Name field.

11. In the New View Name field, type SALES_MARKETING.

12. Click **Apply**.

    The Confirmation window opens.

13. Click **OK**.

**See Also:**

- *Oracle Database SQL Language Reference* for information about the `CREATE VIEW` statement with the `OR REPLACE` clause
- *Oracle Database SQL Language Reference* for information about the `RENAME` statement

## Dropping Views

To drop a view, use either the SQL Developer navigation frame and Drop tool or the DDL statement `DROP VIEW`.

This tutorial shows how to use the navigation frame and Drop tool to drop the view `SALES_MARKETING` (changed in "Tutorial: Changing Views with the Edit View and Rename Tools" on page 6-18). The equivalent DDL statement is:

```
DROP VIEW SALES_MARKETING;
```

**To drop the view SALES_MARKETING using the Drop tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Views**.

   A list of views appears.

3. Right-click **SALES_MARKETING**.

   A list of choices appears.

4. Click **Drop**.

   The Drop window opens.

5. Click **Apply**.

   The Confirmation window opens.

6. Click **OK**.

**See Also:**

- *Oracle Database SQL Language Reference* for information about the `DROP VIEW` statement

# Creating and Managing Sequences

Sequences are schema objects that generate unique sequential values, which are very useful when you need unique primary keys. The `HR` schema has three sequences: `DEPARTMENTS_SEQ`, `EMPLOYEES_SEQ`, and `LOCATIONS_SEQ`.

Sequences are used through the pseudocolumns `CURRVAL` and `NEXTVAL`, which return the current and next values of the sequence, respectively. After creating a sequence, you must initialize it by using `NEXTVAL` to get its first value. Only after the sequence is initialized does `CURRVAL` return its current value.

**Tip:** When you plan to use a sequence to populate the primary key of a table, give the sequence a name that reflects this purpose. (This topic uses the naming convention *`table_name`*`_SEQ`.)

Topics:

- Tutorial: Creating a Sequence
- Dropping Sequences

**See Also:**

- *Oracle Database Concepts* for information about the sequence generator
- *Oracle Database SQL Language Reference* for more information about the CURRVAL and NEXTVAL pseudocolumns
- "Editing Installation Script Files that Create Sequences" on page 10-5

## Tutorial: Creating a Sequence

To create a sequence, use either the SQL Developer tool Create Sequence or the DDL statement CREATE SEQUENCE.

This tutorial shows how to use the Create Database Sequence tool to create a sequence to use to generate primary keys for the EVALUATIONS table (created in "Creating Tables with the CREATE TABLE Statement" on page 6-4). The equivalent DDL statement is:

```
CREATE SEQUENCE EVALUATIONS_SEQ
INCREMENT BY 1
START WITH 1 ORDER;
```

This sequence is part of the sample application that the tutorials and examples in this document show how to develop and deploy.

**To create EVALUATIONS_SEQ using the Create Database Sequence tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Right-click **Sequences**.

   A list of choices appears.

3. Click **New Sequence**.

   The Create Database Sequence window opens. The field Schema has the value HR, and the field Name has the default value, SEQUENCE1.

4. In the Name field, type EVALUATIONS_SEQ over the default value.

5. Click the tab **Properties**.

   The Properties pane appears.

6. In the field Increment, type 1.

7. In the field Start with, type 1.

8. Deselect the check box **Cycle**.

9. Select the check box **Order**.

10. Click **OK**.

    The sequence EVALUATIONS_SEQ is created. To see it, expand Sequences in the navigation frame.

**See Also:**

- *Oracle Database SQL Developer User's Guide* for more information about using SQL Developer to create a sequence
- *Oracle Database SQL Language Reference* for information about the `CREATE SEQUENCE` statement
- "Tutorial: Creating a Trigger that Generates a Primary Key for a Row Before It Is Inserted" on page 8-4 to learn how to create a trigger that inserts the primary keys created by `evaluations_seq` into the `EVALUATIONS` table

## Dropping Sequences

To drop a sequence, use either the SQL Developer navigation frame and Drop tool, or the DDL statement `DROP SEQUENCE`.

**Caution:** Do not drop the sequence that you created in "Tutorial: Creating a Sequence" on page 6-21—you need it for later tutorials. If you want to practice dropping sequences, create new ones and then drop them.

**To drop a sequence using the Drop tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Sequences**.

   A list of sequences appears.

3. Right-click the name of the sequence to drop.

   A list of choices appears.

4. Click **Drop**.

   The Drop window opens.

5. Click **Apply**.

   The Confirmation window opens.

6. Click **OK**.

**See Also:** *Oracle Database SQL Language Reference* for information about the `DROP SEQUENCE` statement

# Creating and Managing Synonyms

A synonym is an alias for another schema object. Some reasons to use synonyms are security (for example, to hide the owner and location of an object) and convenience. Examples of convenience are:

- Using a short synonym, such as `SALES`, for a long object name, such as `ACME_CO.SALES_DATA`
- Using a synonym for a renamed object, instead of changing that object name throughout the applications that use it

For example, if your application uses a table named DEPARTMENTS, and its name changes to DIVISIONS, you can create a DEPARTMENTS synonym for that table and continue to reference it by its original name.

Topics:

- Creating Synonyms
- Dropping Synonyms

**See Also:**

- *Oracle Database Concepts* for additional general information about synonyms

## Creating Synonyms

To create a synonym, use either the SQL Developer tool Create Database Synonym or the DDL statement CREATE SYNONYM.

This tutorial shows how to use the Create Database Synonym tool to create the synonym EMP for the EMPLOYEES table. The equivalent DDL statement is:

```
CREATE SYNONYM EMP FOR EMPLOYEES;
```

This synonym is part of the sample application that the tutorials and examples in this document show how to develop and deploy.

### To create a synonym using the Create Databse Synonym tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Right-click **Synonyms**.

   A list of choices appears.

3. Click **New Synonym**.

   The Create Database Synonym window opens. The check box Public is deselected, the field Schema has the value HR, and the field Name has the default value SYNONYM1.

4. In the Name field, type EMP over the default value.

5. Click the tab **Properties**.

   The Properties pane appears. The Referenced Schema field has the value HR, the option Object based is deselected, and the option Name based is selected.

6. Select **Object Based**.

   The option Name based is now deselected.

7. In the field next to the Object based option, select EMPLOYEES from the drop-down list.

   Object based means that the synonym refers to a specific schema object; in this case, the table EMPLOYEES.

8. Click **OK**.

   The synonym EMP is created. To see it, expand Synonyms in the navigation frame. You can now use EMP instead of EMPLOYEES.

**See Also:** *Oracle Database SQL Language Reference* for information about the `CREATE SYNONYM` statement

## Dropping Synonyms

To drop a synonym, use either the SQL Developer navigation frame and Drop tool, or the DDL statement `DROP SYNONYM`.

**Caution:** Do not drop the synonym that you created in "Creating Synonyms" on page 6-23—you need it for later tutorials. If you want to practice dropping sequences, create new ones and then drop them.

### To drop a synonym using the Drop tool:

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Synonyms**.

   A list of synonyms appears.

3. Right-click the name of the synonym to drop.

   A list of choices appears.

4. Click **Drop**.

   The Drop window opens.

5. Click **Apply**.

   The Confirmation window opens.

6. Click **OK**.

**See Also:** *Oracle Database SQL Language Reference* for information about the `DROP SYNONYM` statement

# 7

# Developing Stored Subprograms and Packages

This chapter contains the following topics:



> **Tip:** If you have problems creating or running PL/SQL code, check the Oracle Database trace files. The `USER_DUMP_DEST` initialization parameter specifies the current location of the trace files. You can find the value of this parameter by issuing `SHOW PARAMETER USER_DUMP_DEST` in the SQL Worksheet of SQL Developer or in SQL*Plus. For more information about trace files, see *Oracle Database Performance Tuning Guide*.

## About Stored Subprograms

A **subprogram** is a PL/SQL unit that consists of SQL and PL/SQL statements that solve a specific problem or perform a set of related tasks. A subprogram can have parameters, whose values are supplied by the invoker. A subprogram can be either a procedure or a function. Typically, you use a procedure to perform an action and a function to compute and return a value.

A **stored subprogram** is a subprogram that is stored in the database. Because they are stored in the database, stored programs can be used as building blocks for many different database applications. (A subprogram that is declared within another subprogram, or within an anonymous block, is called a **nested subprogram** or **local**

**subprogram**. It cannot be invoked from outside the subprogram or block in which it is declared. An **anonymous block** is a block that is not stored in the database.)

There are two kinds of stored subprograms:

- **Standalone stored subprogram**, which is created at schema level
- **Packaged subprogram**, which is created inside a package

Standalone stored subprograms are useful for testing pieces of program logic, but when you are sure that they work as intended, Oracle recommends that you put them into packages.

**See Also:**

- *Oracle Database Concepts* for general information about stored subprograms
- *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL subprograms

## About Packages

A **package** is a PL/SQL unit that consists of related subprograms and the explicit cursors and variables that they use.

Oracle recommends that you put your subprograms into packages. Some of the reasons are:

- Packages allow you to hide implementation details from client programs.

  Hiding implementation details from client programs is a widely accepted best practice. Many Oracle customers follow this practice strictly, allowing client programs to access the database only by calling PL/SQL subprograms. Some customers allow client programs to use `SELECT` statements to retrieve information from database tables, but require them to call PL/SQL subprograms for all business functions that change the database.

- Packaged subprograms must be qualified with package names when invoked, which ensures that their names will always work.

  For example, suppose that you developed a schema-level procedure named `CONTINUE` before Oracle Database 11*g* Release 1 (11.1). Release 11.1 introduced the `CONTINUE` statement. Therefore, if you ported your code to 11.1, it would no longer compile. However, if you had developed your procedure inside a package, your code would refer to the procedure as *`package_name`*`.CONTINUE`, so the code would still compile.

- Packaged subprograms can send and receive records and collections.

  Standalone stored subprograms can send and receive only scalar parameters—single values with no internal components, such as `VARCHAR2`, `NUMBER`, and `DATE`.

**Note:** Oracle Database supplies many PL/SQL packages to extend database functionality and provide PL/SQL access to SQL features. You can use the supplied packages when creating your applications or for ideas in creating your own stored procedures. For information about these packages, see *Oracle Database PL/SQL Packages and Types Reference*.

**See Also:**

- *Oracle Database Concepts* for general information about packages
- *Oracle Database PL/SQL Language Reference* for more reasons to use packages
- *Oracle Database PL/SQL Language Reference* for complete information about PL/SQL packages
- *Oracle Database PL/SQL Packages and Types Reference* for complete information about the PL/SQL packages that Oracle provides

## About PL/SQL Identifiers

Every PL/SQL subprogram, package, parameter, variable, constant, exception, and explicit cursor has a name, which is a PL/SQL identifier.

The minimum length of an identifier is one character; the maximum length is 30 characters. The first character must be a letter, but each later character can be either a letter, numeral, dollar sign ($), underscore (_), or number sign (#). For example, these are acceptable identifiers:

```
X
t2
phone#
credit_limit
LastName
oracle$number
money$$$tree
SN##
try_again_
```

PL/SQL is not case-sensitive for identifiers. For example, PL/SQL considers these to be the same:

```
lastname
LastName
LASTNAME
```

You cannot use a PL/SQL reserved word as an identifier. You can use a PL/SQL keyword as an identifier, but it is not recommended. For lists of PL/SQL reserved words and keywords, see *Oracle Database PL/SQL Language Reference*.

**Tip:** Use meaningful names for identifiers, and follow a naming convention. For example, start each constant name with `con_`, each variable name with `var_`, and so on.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for additional general information about PL/SQL identifiers
- *Oracle Database PL/SQL Language Reference* for additional information about PL/SQL naming conventions
- *Oracle Database PL/SQL Language Reference* for information about the scope and visibility of PL/SQL identifiers
- *Oracle Database PL/SQL Language Reference* for information how to collect data on PL/SQL identifiers
- *Oracle Database PL/SQL Language Reference* for information about how PL/SQL resolves identifier names

## About PL/SQL Data Types

Every PL/SQL constant, variable, subprogram parameter, and function return value has a data type that determines its storage format, constraints, valid range of values, and operations that can be performed on it.

A PL/SQL data type is either a SQL data type (such as `VARCHAR2`, `NUMBER`, and `DATE`) or a PL/SQL-only data type. The latter include `BOOLEAN`, `RECORD`, `REF CURSOR`, and many predefined subtypes. PL/SQL also lets you define your own subtypes.

A **subtype** is a subset of another data type, which is called its **base type**. A subtype has the same valid operations as its base type, but only a subset of its valid values. Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables.

The predefined numeric subtype `PLS_INTEGER` is especially useful, because its operations use hardware arithmetic, rather than the library arithmetic that its base type uses.

You cannot use PL/SQL-only data types at schema level (that is, in tables or standalone stored subprograms). Therefore, to use them in a stored subprogram, you must put the subprogram in a package.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for general information about PL/SQL data types
- *Oracle Database PL/SQL Language Reference* for information about the `PLS_INTEGER` data type
- "About SQL Data Types" on page 6-2

## Creating and Managing Standalone Stored Subprograms

Topics:

- Dropping Standalone Stored Subprograms

> **Note:** To do the tutorials in this document, you must be connected to Oracle Database as the user HR from SQL Developer. For instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6.

## About Subprogram Structure

A subprogram follows PL/SQL block structure; that is, it has:

- **Declarative part** (optional)

  The declarative part contains declarations of types, constants, variables, exceptions, explicit cursors, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

- **Executable part** (required)

  The executable part contains statements that assign values, control execution, and manipulate data.

- **Exception-handling part** (optional)

  The exception-handling part contains code that handles exceptions (run-time errors).

**Comments** can appear anywhere in PL/SQL code. The PL/SQL compiler ignores them. Adding comments to your program promotes readability and aids understanding. A **single-line comment** starts with a double hyphen (`--`) and extends to the end of the line. A **multiline comment** starts with a slash and asterisk (`/*`) and ends with an asterisk and a slash (`*/`).

The structure of a procedure is:

```
PROCEDURE name [ ( parameter_list ) ]
{ IS | AS }
  [ declarative_part ]
BEGIN  -- executable part begins
  statement; [ statement; ]...
[ EXCEPTION -- executable part ends, exception-handling part begins]
  exception_handler; [ exception_handler; ]... ]
END; /* exception-handling part ends if it exists;
        otherwise, executable part ends */
```

The structure of a function is like that of a procedure, except that it includes a `RETURN` clause and at least one `RETURN` statement (and some optional clauses that are beyond the scope of this document):

```
FUNCTION name [ ( parameter_list ) ] RETURN data_type [ clauses ]
{ IS | AS }
  [ declarative_part ]
BEGIN  -- executable part begins
  -- at least one statement must be a RETURN statement
  statement; [ statement; ]...
[ EXCEPTION -- executable part ends, exception-handling part begins]
  exception_handler; [ exception_handler; ]... ]
END; /* exception-handling part ends if it exists;
        otherwise, executable part ends */
```

The code that begins with PROCEDURE or FUNCTION and ends before IS or AS is the **subprogram signature**. The declarative, executable, and exception-handling parts comprise the **subprogram body**. The syntax of *exception-handler* is in "About Exceptions and Exception Handlers" on page 7-48.

> **See Also:** *Oracle Database PL/SQL Language Reference* for more information about subprogram parts

## Tutorial: Creating a Standalone Stored Procedure

To create a standalone stored procedure, use either the SQL Developer tool Create PL/SQL Procedure or the DDL statement CREATE PROCEDURE.

This tutorial shows how to use the Create PL/SQL Procedure tool to create a standalone stored procedure named ADD_EVALUATION that adds a row to the EVALUATIONS table (created in "Creating Tables with the CREATE TABLE Statement" on page 6-4).

**To create a standalone stored procedure using Create PL/SQL Procedure tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Right-click **Procedures**.

   A list of choices appears.

3. Click **New Procedure**.

   The Create PL/SQL Procedure window opens.

4. For Schema, accept the default value, HR.

5. For Name, change PROCEDURE1 to ADD_EVALUATION.

6. Click the icon **Add Column**.

   A row appears under the column headings. Its fields have these default values: Name, param1; Type, VARCHAR2; Mode, IN; Default Value, empty.

7. For Name, change param1 to evaluation_id.

8. For Type, select NUMBER from the drop-down list.

9. For Mode, accept the default value, IN.

10. Leave Default Value empty.

11. Add a second parameter by repeating steps 6 through 10 with the Name employee_id and the Type NUMBER.

12. Add a third parameter by repeating steps 6 through 10 with the Name evaluation_date and the Type DATE.

13. Add a fourth parameter by repeating steps 6 through 10 with the Name job_id and the Type VARCHAR2.

14. Add a fifth parameter by repeating steps 6 through 10 with the Name manager_id and the Type NUMBER.

15. Add a sixth parameter by repeating steps 6 through 10 with the Name department_id and the Type NUMBER.

16. Add a seventh parameter by repeating steps 6 through 10 with the Name total_score and the Type NUMBER.

17. Click **OK**.

    The ADD_EVALUATION pane opens, showing the CREATE PROCEDURE statement that created the procedure:

```
CREATE OR REPLACE
PROCEDURE ADD_EVALUATION
( evaluation_id IN NUMBER
, employee_id IN NUMBER
, evaluation_date IN DATE
, job_id IN VARCHAR2
, manager_id IN NUMBER
, department_id IN NUMBER
, total_score IN NUMBER
) AS
BEGIN
  NULL;
END ADD_EVALUATION;
```

    Because the only statement in the execution part of the procedure is NULL, the procedure does nothing.

18. Replace NULL with this statement:

```
INSERT INTO EVALUATIONS (
   evaluation_id,
   employee_id,
   evaluation_date,
   job_id,
   manager_id,
   department_id,
   total_score
)
VALUES (
  ADD_EVALUATION.evaluation_id,
  ADD_EVALUATION.employee_id,
  ADD_EVALUATION.evaluation_date,
  ADD_EVALUATION.job_id,
  ADD_EVALUATION.manager_id,
  ADD_EVALUATION.department_id,
  ADD_EVALUATION.total_score
);
```

    (Qualifying the parameter names with the procedure name ensures that they are not confused with the columns that have the same names.)

    The title of the ADD_EVALUATION pane is in italic font, indicating that the procedure is not yet saved in the database.

19. From the File menu, select **Save**.

    Oracle Database compiles the procedure and saves it. The title of the ADD_EVALUATION pane is no longer in italic font.

**See Also:**

- *Oracle Database SQL Developer User's Guide* for another example of using SQL Developer to create a standalone stored procedure
- "About Data Definition Language (DDL) Statements" on page 6-1 for general information that applies to the `CREATE PROCEDURE` statement
- *Oracle Database PL/SQL Language Reference* for information about the `CREATE PROCEDURE` statement

## Tutorial: Creating a Standalone Stored Function

To create a standalone stored function, use either the SQL Developer tool Create PL/SQL Function or the DDL statement `CREATE FUNCTION`.

This tutorial shows how to use the Create PL/SQL Function tool to create a standalone stored function named `calculate_score` that has three parameters and returns a value of type `NUMBER`.

**To create a standalone stored function using Create PL/SQL Function tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Right-click **Functions**.

   A list of choices appears.

3. Click **New Function**.

   The Create PL/SQL Function window opens. It looks like the Create PL/SQL Procedure window (see "Tutorial: Creating a Standalone Stored Procedure" on page 7-6) except that its Parameters pane has a row for the value that the function returns. In that row, the value of Name is `<Return>` and the default value of Type is `VARCHAR2`.

4. For Schema, accept the default value, `HR`.
5. For Name, change `FUNCTION1` to `calculate_score`.
6. In the Parameters pane, in the Type field of the only row, select `NUMBER` from the drop-down list.
7. Click the icon **Add Column**.

   A row appears under the column headings. Its fields have these default values: Name, `param1`; Type, `VARCHAR2`; Mode, `IN`; Default Value, empty.

8. For Name, change `param1` to `cat`.
9. For Type, accept the default, `VARCHAR2`.
10. For Mode, accept the default value, `IN`.
11. Leave Default Value empty.
12. Add a second parameter by repeating steps 7 through 11 with the Name `score` and the Type `NUMBER`.
13. Add a third parameter by repeating steps 7 through 11 with the Name `weight` and the Type `NUMBER`.
14. Click **OK**.

The calculate_score pane opens, showing the CREATE FUNCTION statement that created the function:

```
CREATE OR REPLACE
FUNCTION calculate_score
( cat IN VARCHAR2
, score IN NUMBER
, weight IN NUMBER
) RETURN NUMBER AS
BEGIN
  RETURN NULL;
END calculate_score;
```

Because the only statement in the execution part of the function is RETURN NULL, the function does nothing.

15. Replace NULL with score * weight.

    The title of the calculate_score pane is in italic font, indicating that the function is not yet saved in the database.

16. Select **Save** from the File menu.

    Oracle Database compiles the function and saves it. The title of the calculate_score pane is no longer in italic font.

**See Also:**

- "About Data Definition Language (DDL) Statements" on page 6-1 for general information that applies to the CREATE FUNCTION statement
- *Oracle Database PL/SQL Language Reference* for information about the CREATE FUNCTION statement

## Changing Standalone Stored Subprograms

To change a standalone stored subprogram, use either the SQL Developer tool Edit or the DDL statement ALTER PROCEDURE or ALTER FUNCTION.

### To change a standalone stored subprogram using the Edit tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand either **Functions** or **Procedures**.

   A list of functions or procedures appears.

3. Click the function or procedure to change.

   To the right of the Connections pane, a frame appears. Its top tab has the name of the subprogram to change. Under the top tab are subtabs.

4. Click the subtab **Code**.

   The Code pane appears, showing the code that created the subprogram to change.

5. Click the icon **Edit**.

   Another pane appears, also with the name of the subprogram to change.

6. In the pane, change the code.

The title of the pane changes to italic font, indicating that the change is not yet saved in the database.

7. Select **Save** from the File menu.

   Oracle Database compiles the subprogram and saves it. The title of the pane is no longer in italic font.

   **See Also:**

   - "About Data Definition Language (DDL) Statements" on page 6-1 for general information that applies to the `ALTER PROCEDURE` and `ALTER FUNCTION` statements
   - *Oracle Database PL/SQL Language Reference* for information about the `ALTER PROCEDURE` statement
   - *Oracle Database PL/SQL Language Reference* for information about the `ALTER FUNCTION` statement

## Tutorial: Testing a Standalone Stored Function

This tutorial shows how to use the SQL Developer tool Run to test the standalone stored function `calculate_score` (created in "Tutorial: Creating a Standalone Stored Function" on page 7-8).

### To test the calculate_score function using the Run tool:

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Functions**.

   A list of functions appears.

3. Right-click **calculate_score**.

   A list of choices appears.

4. Click **Run**.

   The Run PL/SQL window opens. Its PL/SQL Block frame includes this code:

```
v_Return := calculate_score (
    CAT => CAT,
    SCORE => SCORE,
    WEIGHT => WEIGHT
  );
```

5. Change the values of `SCORE` and `WEIGHT` to 8 and 0.2, respectively:

```
v_Return := calculate_score (
    CAT => CAT,
    SCORE => 8,
    WEIGHT => 0.2
  );
```

6. Click **OK**.

   The Running - Log window opens, showing this result:

```
Connecting to the database hr_conn.
v_Return = 1.6
Process exited.
Disconnecting from the database hr_conn.
```

**See Also:** *Oracle Database SQL Developer User's Guide* for information about using SQL Developer to run and debug procedures and functions

## Dropping Standalone Stored Subprograms

To drop a standalone stored subprogram, use either the SQL Developer navigation frame and Drop tool, or the DDL statement `DROP PROCEDURE` or `DROP FUNCTION`.

**To drop a standalone stored subprogram using the Drop tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand either **Functions** or **Procedures**.

   A list of functions or procedures appears.

3. Right-click the name of the function or procedure to drop.

   A list of choices appears.

4. Click **Drop**.

   The Drop window opens.

5. Click **Apply**.

   The Confirmation window opens.

6. Click **OK**.

**See Also:**

- "About Data Definition Language (DDL) Statements" on page 6-1 for general information that applies to the `DROP PROCEDURE` and `DROP FUNCTION` statements
- *Oracle Database SQL Language Reference* for information about the `DROP PROCEDURE` statement
- *Oracle Database SQL Language Reference* for information about the `DROP FUNCTION` statement

# Creating and Managing Packages

Topics:

- About Package Structure
- Tutorial: Creating a Package Specification
- Tutorial: Changing a Package Specification
- Tutorial: Creating a Package Body
- Dropping a Package

**See Also:** "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16, which shows how to change a package body

## About Package Structure

A package always has a specification, and it usually has a body.

The **package specification** defines the package, declaring the types, variables, constants, exceptions, explicit cursors, and subprograms that can be referenced from outside the package. A package specification is an **application program interface (API)**: It has all the information that client programs need to call its subprograms, but no information about their implementation.

The **package body** defines the queries for the explicit cursors, and the code for the subprograms, that are declared in the package specification (therefore, a package with neither explicit cursors nor subprograms does not need a body). The package body can also define **local subprograms**, which are not declared in the specification and can be invoked only by other subprograms in the package. Package body contents are hidden from client programs. You can change the package body without invalidating the applications that call the package.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about the package specification
- *Oracle Database PL/SQL Language Reference* for more information about the package body

## Tutorial: Creating a Package Specification

To create a package specification, use either the SQL Developer tool Create PL/SQL Package or the DDL statement `CREATE PACKAGE`.

This tutorial shows how to use the Create PL/SQL Package tool to create a specification for a package named `EMP_EVAL`.

This package specification is the API for the sample application that the tutorials and examples in this document show how to develop and deploy.

**To create a package specification using Create PL/SQL Package tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Right-click **Packages**.

   A list of choices appears.

3. Click **New Package**.

   The Create PL/SQL Package window opens. The field Schema has the value `HR`, the field Name has the default value `PACKAGE1`, and the check box Add New Source In Lowercase is deselected.

4. For Schema, accept the default value, `HR`.

5. For Name, change `PACKAGE1` to `EMP_EVAL`.

6. Click **OK**.

   The `EMP_EVAL` pane opens, showing the `CREATE PACKAGE` statement that created the package:

```
CREATE OR REPLACE PACKAGE emp_eval AS

  /* TODO enter package declarations (types, exceptions, methods etc) here */
```

```
END emp_eval;
```

The title of the pane is in italic font, which indicates that the package is not saved to the database.

7. (Optional) In the `CREATE PACKAGE` statement, replace the comment with declarations.

   If you do not do this step now, you can do it later, as shown in "Tutorial: Changing a Package Specification" on page 7-13.

8. From the File menu, select **Save**.

   Oracle Database compiles the package and saves it. The title of the `EMP_EVAL` pane is no longer in italic font.

   **See Also:** *Oracle Database PL/SQL Language Reference* for information about the `CREATE PACKAGE` statement (for the package specification)

## Tutorial: Changing a Package Specification

To change a package specification, use either the SQL Developer tool Edit or the DDL statement `CREATE PACKAGE` with the `OR REPLACE` clause.

This tutorial shows how to use the Edit tool to change the specification for the `EMP_EVAL` package (created in "Tutorial: Creating a Package Specification" on page 7-12). Specifically, the tutorial shows how to add declarations for a procedure, `eval_department`, and a function, `calculate_score`.

### To change EMP_EVAL package specification using the Edit tool:

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Packages**.

   A list of packages appears.

3. Right-click **EMP_EVAL**.

   A list of choices appears.

4. Click **Edit**.

   The `EMP_EVAL` pane opens, showing the `CREATE PACKAGE` statement that created the package:

```
CREATE OR REPLACE PACKAGE emp_eval AS

  /* TODO enter package declarations (types, exceptions, methods etc) here */

END emp_eval;
```

   The title of the pane is not in italic font, which indicates that the package is saved in the database.

5. In the `EMP_EVAL` pane, replace the comment with this code:

```
PROCEDURE eval_department ( dept_id IN NUMBER );

FUNCTION calculate_score ( evaluation_id IN NUMBER
                         , performance_id IN NUMBER)
                         RETURN NUMBER;
```

A new EMP_EVAL pane opens, showing the changed CREATE PACKAGE statement. The title of the pane is in italic font, which indicates that the changes have not been saved to the database.

6. Click the icon **Compile**.

   The changed package specification compiles and is saved to the database. The title of the EMP_EVAL pane is no longer in italic font.

   **See Also:** *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE statement with the OR REPLACE clause

## Tutorial: Creating a Package Body

To create a package body, use either the SQL Developer tool Create Body or the DDL statement CREATE PACKAGE BODY.

This tutorial shows how to use the Create Body tool to create a body for the EMP_EVAL package (created in "Tutorial: Creating a Package Specification" on page 7-12 and changed in "Tutorial: Changing a Package Specification" on page 7-13).

This package body will contain the implementation details of the sample application that the tutorials and examples in this document show how to develop and deploy.

### To create a body for the package EMP_EVAL using the Create Body tool:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Packages**.

   A list of packages appears.

3. Right-click **EMP_EVAL**.

   A list of choices appears.

4. Click **Create Body**.

   The EMP_EVAL Body pane appears, showing the automatically generated code for the package body:

```
CREATE OR REPLACE
PACKAGE BODY emp_eval AS

  PROCEDURE eval_department(dept_id IN NUMBER) AS
  BEGIN
    /* TODO implementation required */
    NULL;
  END eval_department;

  FUNCTION calculate_score ( evaluation_id IN NUMBER
                           , performance_id IN NUMBER)
                           RETURN NUMBER AS
  BEGIN
    /* TODO implementation required */
    RETURN NULL;
  END calculate_score;

END emp_eval;
```

The title of the pane is in italic font, which indicates that the code is not saved in the database.

5. (Optional) In the `CREATE PACKAGE BODY` statement:
   - Replace the comments with executable statements.
   - (Optional) In the executable part of the procedure, either delete `NULL` or replace it with an executable statement.
   - (Optional) In the executable part of the function, either replace `NULL` with another expression.

   If you do not do this step now, you can do it later, as shown in "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16.
6. Click the icon **Compile**.

   The changed package body compiles and is saved to the database. The title of the `EMP_EVAL` Body pane is no longer in italic font.

> **See Also:** *Oracle Database PL/SQL Language Reference* for information about the `CREATE PACKAGE BODY` statement (for the package body)

## Dropping a Package

To drop a package (specification and body), use either the SQL Developer navigation frame and Drop tool, or the DDL statement `DROP PACKAGE`.

**To drop a package using the Drop tool:**

1. On the Connections tab, expand the `hr_conn` information by clicking the plus sign (+) to the left of the `hr_conn` icon.

   Under the `hr_conn` icon, a list of schema object types appears.
2. Expand **Packages**.

   A list of packages appears.
3. Right-click the name of the package to drop.

   A list of choices appears.
4. Click **Drop Package**.

   The Drop window opens.
5. Click **Apply**.

   The Confirmation window opens.
6. Click **OK**.

> **See Also:** *Oracle Database PL/SQL Language Reference* for information about the `DROP PACKAGE` statement

# Declaring and Assigning Values to Variables and Constants

One significant advantage that PL/SQL has over SQL is that PL/SQL lets you declare and use variables and constants.

A variable or constant declared in a package specification can be used by any program that has access to the package. A variable or constant declared in a package body or subprogram is local to that package or subprogram.

A **variable** holds a value of a particular data type. Your program can change the value at run time. A **constant** holds a value that cannot be changed.

A variable or constant can have any PL/SQL data type. When declaring a variable, you can assign it an initial value; if you do not, its initial value is `NULL`. When declaring a constant, you must assign it an initial value. To assign an initial value to a variable or constant, use the assignment operator (`:=`).

> **Tip:** Declare all values that do not change as constants. This practice optimizes your compiled code and makes your source code easier to maintain.

Topics:

- Tutorial: Declaring Variables and Constants in a Subprogram
- Ensuring that Variables, Constants, and Parameters Have Correct Data Types
- Tutorial: Changing Declarations to Use the %TYPE Attribute
- Assigning Values to Variables

> **See Also:** *Oracle Database PL/SQL Language Reference* for general information about variables and constants

## Tutorial: Declaring Variables and Constants in a Subprogram

This tutorial shows how to use the SQL Developer tool Edit to declare variables and constants in the `EMP_EVAL.calculate_score` function (specified in "Tutorial: Creating a Package Specification" on page 7-12). (This tutorial is also an example of changing a package body.)

### To declare variables and constants in calculate_score function:

1. On the Connections tab, expand the `hr_conn` information by clicking the plus sign (**+**) to the left of the `hr_conn` icon.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Packages**.

   A list of packages appears.

3. Expand **EMP_EVAL**.

   A list appears.

4. Right-click **EMP_EVAL Body**.

   A list of choices appears.

5. Click **Edit**.

   The `EMP_EVAL` Body pane appears, showing the code for the package body:

```
CREATE OR REPLACE
PACKAGE BODY emp_eval AS

  PROCEDURE eval_department ( dept_id IN NUMBER ) AS

  BEGIN
    /* TODO implementation required */
    NULL;
  END eval_department;
```

```
  FUNCTION calculate_score ( evaluation_id IN NUMBER
                           , performance_id IN NUMBER)
                           RETURN NUMBER AS
  BEGIN
    /* TODO implementation required */
    RETURN NULL;
  END calculate_score;

END emp_eval;
```

6. Between `RETURN NUMBER AS` and `BEGIN`, add these variable and constant declarations:

```
n_score       NUMBER(1,0);               -- variable
n_weight      NUMBER;                    -- variable
max_score     CONSTANT NUMBER(1,0) := 9;  -- constant, initial value 9
max_weight    CONSTANT NUMBER(8,8) := 1;  -- constant, initial value 1
```

7. From the File menu, select **Save**.

   Oracle Database saves the changed package body.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for general information about declaring variables and constants
- "Assigning Values to Variables with the Assignment Operator" on page 7-20

## Ensuring that Variables, Constants, and Parameters Have Correct Data Types

After "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16, the code for the `calculate_score` function, in the body of the package `EMP_EVAL`, is:

```
FUNCTION calculate_score ( evaluation_id IN NUMBER
                         , performance_id IN NUMBER )
                         RETURN NUMBER AS
  n_score       NUMBER(1,0);               -- variable
  n_weight      NUMBER;                    -- variable
  max_score     CONSTANT NUMBER(1,0) := 9;  -- constant, initial value 9
  max_weight    CONSTANT NUMBER(8,8) := 1;  -- constant, initial value 1
  BEGIN
    /* TODO implementation required */
    RETURN NULL;
  END calculate_score;
```

The variables, constants, and parameters of the function represent values from the tables `SCORES` (created in "Creating Tables with the CREATE TABLE Statement" on page 6-4) and `PERFORMANCE_PARTS` (created in "Tutorial: Creating a Table with the Create Table Tool" on page 6-3):

- Variable `n_score` will hold a value from the `SCORE` column of the `SCORES` table, and constant `max_score` will be compared to such values.
- Variable `n_weight` will hold a value from the `WEIGHT` column of the `PERFORMANCE_PARTS` table, and constant `max_weight` will be compared to such values.
- Parameter `evaluation_id` will hold a value from the `EVALUATION_ID` column of the `SCORES` table.

- Parameter `performance_id` will hold a value from the `PERFORMANCE_ID` column of the `SCORES` table.

Therefore, each variable, constant, and parameter has the same data type as its corresponding column.

If the data types of the columns change, you want the data types of the variables, constants, and parameters to change to the same data types; otherwise, the `calculate_score` function will be invalidated.

To ensure that the data types of the variables, constants, and parameters will always match those of the columns, declare them with the `%TYPE` attribute. The `%TYPE` attribute supplies the data type of a table column or another variable, ensuring the correct data type assignment.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about the `%TYPE` attribute
- *Oracle Database PL/SQL Language Reference* for the syntax of the `%TYPE` attribute

## Tutorial: Changing Declarations to Use the %TYPE Attribute

This tutorial shows how to use the SQL Developer tool Edit to change the declarations of the variables, constants, and formal parameters of the `EMP_EVAL.calculate_score` function (shown in "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16) to declarations that use the `%TYPE` attribute.

**To change the declarations in calculate_score to use %TYPE:**

1. On the Connections tab, expand the `hr_conn` information by clicking the plus sign (+) to the left of the `hr_conn` icon.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Packages**.

   A list of packages appears.

3. Expand **EMP_EVAL**.

   A list appears.

4. Right-click **EMP_EVAL Body**.

   A list of choices appears.

5. Click **Edit**.

   The `EMP_EVAL` Body pane appears, showing the code for the package body:

```
CREATE OR REPLACE
PACKAGE BODY emp_eval AS

  PROCEDURE eval_department ( dept_id IN NUMBER ) AS
  BEGIN
    /* TODO implementation required */
    NULL;
  END eval_department;

  FUNCTION calculate_score ( evaluation_id IN NUMBER
                           , performance_id IN NUMBER )
                           RETURN NUMBER AS
```

```
  n_score       NUMBER(1,0);               -- variable
  n_weight      NUMBER;                    -- variable
  max_score     CONSTANT NUMBER(1,0) := 9;  -- constant, initial value 9
  max_weight    CONSTANT NUMBER(8,8) := 1;  -- constant, initial value 1
  BEGIN
    /* TODO implementation required */
    RETURN NULL;
  END calculate_score;

END emp_eval;
```

6. In the code for the function, make the changes shown in bold font:

```
  FUNCTION calculate_score ( evaluation_id IN SCORES.EVALUATION_ID%TYPE
                           , performance_id IN SCORES.PERFORMANCE_ID%TYPE)
                           RETURN NUMBER AS
  n_score       SCORES.SCORE%TYPE;
  n_weight      PERFORMANCE_PARTS.WEIGHT%TYPE;
  max_score     CONSTANT SCORES.SCORE%TYPE := 9;
  max_weight    CONSTANT PERFORMANCE_PARTS.WEIGHT%TYPE := 1;
```

7. Right-click **EMP_EVAL**.

   A list of choices appears.

8. Click **Edit**.

   The EMP_EVAL pane opens, showing the CREATE PACKAGE statement that created the package:

```
CREATE OR REPLACE PACKAGE emp_eval AS

PROCEDURE eval_department(dept_id IN NUMBER);
FUNCTION calculate_score(evaluation_id IN NUMBER
                       , performance_id IN NUMBER)
                         RETURN NUMBER;

END emp_eval;
```

9. In the code for the function, make the changes shown in bold font:

```
FUNCTION calculate_score(evaluation_id IN scores.evaluation_id%TYPE
                       , performance_id IN scores.performance_id%TYPE)
```

10. Right-click **EMP_EVAL**.

    A list of choices appears.

11. Click **Compile**.

## Assigning Values to Variables

You can assign a value to a variable in these ways:

- Use the assignment operator to assign it the value of an expression.
- Use the SELECT INTO or FETCH statement to assign it a value from a table.
- Pass it to a subprogram as an OUT or IN OUT parameter, and then assign the value inside the subprogram
- Bind the variable to a value.

Topics:

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about assigning values to variables
- *Oracle Database 2 Day + .NET Developer's Guide* for information about binding variables
- *Oracle Database 2 Day + PHP Developer's Guide* for information about binding variables
- *Oracle Database 2 Day + Java Developer's Guide* for information about binding variables
- *Oracle Database 2 Day + Application Express Developer's Guide* for information about binding variables

### Assigning Values to Variables with the Assignment Operator

With the assignment operator (`:=`), you can assign the value of an expression to a variable in either the declarative or executable part of a subprogram.

In the declarative part of a subprogram, you can assign an initial value to a variable when you declare it. The syntax is:

```
variable_name data_type := expression;
```

In the executable part of a subprogram, you can assign a value to a variable with an assignment statement. The syntax is:

```
variable_name := expression;
```

Example 7–1 shows, in bold font, the changes to make to the `EMP_EVAL.calculate_score` function to add a variable, `running_total`, and use it as the return value of the function. The assignment operator appears in both the declarative and executable parts of the function. (The data type of `running_total` must be `NUMBER`, rather than `SCORES.SCORE%TYPE` or `PERFORMANCE_PARTS.WEIGHT%TYPE`, because it holds the product of two `NUMBER` values with different precisions and scales.)

***Example 7–1 Assigning Values to a Variable with Assignment Operator***

```
FUNCTION calculate_score(evaluation_id IN SCORES.EVALUATION_ID%TYPE
                         , performance_id IN SCORES.PERFORMANCE_ID%TYPE)
                         RETURN NUMBER AS
  n_score       SCORES.SCORE%TYPE;
  n_weight      PERFORMANCE_PARTS.WEIGHT%TYPE;
  running_total NUMBER := 0;
  max_score     CONSTANT SCORES.SCORE%TYPE := 9;
  max_weight    CONSTANT PERFORMANCE_PARTS.WEIGHT%TYPE:= 1;
BEGIN
  running_total := max_score * max_weight;
  RETURN running_total;
END calculate_score;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for variable declaration syntax
- *Oracle Database PL/SQL Language Reference* for assignment statement syntax

### Assigning Values to Variables with the SELECT INTO Statement

To use table values in subprograms or packages, you must assign them to variables with `SELECT INTO` statements.

Example 7–2 shows, in bold font, the changes to make to the `EMP_EVAL.calculate_score` function to have it calculate `running_total` from table values.

***Example 7–2 Assigning Table Values to Variables with SELECT INTO***

```
FUNCTION calculate_score ( evaluation_id IN scores.evaluation_id%TYPE
                         , performance_id IN scores.performance_id%TYPE )
                         RETURN NUMBER AS

  n_score       scores.score%TYPE;
  n_weight      performance_parts.weight%TYPE;
  running_total NUMBER := 0;
  max_score     CONSTANT scores.score%TYPE := 9;
  max_weight    CONSTANT performance_parts.weight%TYPE:= 1;
BEGIN
  SELECT s.score INTO n_score
  FROM SCORES s
  WHERE evaluation_id = s.evaluation_id
  AND performance_id = s.performance_id;

  SELECT p.weight INTO n_weight
  FROM PERFORMANCE_PARTS p
  WHERE performance_id = p.performance_id;

  running_total := n_score * n_weight;
  RETURN running_total;
END calculate_score;
```

The `add_eval` procedure in Example 7–3 inserts a row into the `EVALUATIONS` table, using values from the corresponding row in the `EMPLOYEES` table. Add the `add_eval` procedure to the body of the `EMP_EVAL` package, but not to the specification. Because it is not in the specification, `add_eval` is local to the package—it can be invoked only by other subprograms in the package, not from outside the package.

***Example 7–3 Inserting a Table Row with Values from Another Table***

```
PROCEDURE add_eval ( employee_id IN EMPLOYEES.EMPLOYEE_ID%TYPE
                   , today IN DATE )
AS
  job_id         EMPLOYEES.JOB_ID%TYPE;
  manager_id     EMPLOYEES.MANAGER_ID%TYPE;
  department_id  EMPLOYEES.DEPARTMENT_ID%TYPE;
BEGIN
  SELECT e.job_id INTO job_id
  FROM EMPLOYEES e
  WHERE employee_id = e.employee_id;

  SELECT e.manager_id INTO manager_id
```

```
  FROM EMPLOYEES e
  WHERE employee_id = e.employee_id;

  SELECT e.department_id INTO department_id
  FROM EMPLOYEES e
  WHERE employee_id = e.employee_id;

  INSERT INTO EVALUATIONS (
    evaluation_id,
    employee_id,
    evaluation_date,
    job_id,
    manager_id,
    department_id,
    total_score
  )
  VALUES (
    evaluations_seq.NEXTVAL,    -- evaluation_id
    add_eval.employee_id,       -- employee_id
    add_eval.today,             -- evaluation_date
    add_eval.job_id,            -- job_id
    add_eval.manager_id,        -- manager_id
    add_eval.department_id,     -- department_id
    0                           -- total_score
  );
END add_eval;
```

**See Also:** *Oracle Database PL/SQL Language Reference* for more information about the `SELECT INTO` statement

# Controlling Program Flow

Unlike SQL, which runs statements in the order in which you enter them, PL/SQL has control statements that let you control the flow of your program.

Topics:

- About Control Statements
- Using the IF Statement
- Using the CASE Statement
- Using the FOR LOOP Statement
- Using the WHILE LOOP Statement
- Using the Basic LOOP and EXIT WHEN Statements

## About Control Statements

PL/SQL has three categories of control statements:

- **Conditional selection statements**, which let you execute different statements for different data values.

  The conditional selection statements are `IF` and and `CASE`.

- **Loop statements**, which let you repeat the same statements with a series of different data values.

  The loop statements are `FOR LOOP`, `WHILE LOOP` and basic `LOOP`.

The EXIT statement transfers control to the end of a loop. The CONTINUE statement exits the current iteration of a loop and transfers control to the next iteration. Both EXIT and CONTINUE have an optional WHEN clause, in which you can specify a condition.

- **Sequential control statements**, which let you go to a specified, labeled statement, or to do nothing.

  The sequential control statements are GOTO and and NULL.

  **See Also:**

  - *Oracle Database PL/SQL Language Reference* for an overview of PL/SQL control statements

## Using the IF Statement

The IF statement either executes or skips a sequence of statements, depending on the value of a Boolean expression.

The IF statement has this syntax:

```
IF boolean_expression THEN statement [, statement ]
[ ELSEIF boolean_expression THEN statement [, statement ] ]...
[ ELSE  statement [, statement ] ]
END IF;
```

Suppose that your company evaluates employees twice a year in the first 10 years of employment, but only once a year afterward. You want a function that returns the evaluation frequency for an employee. You can use an IF statement to determine the return value of the function, as in Example 7–4.

Add the eval_frequency function to the body of the EMP_EVAL package, but not to the specification. Because it is not in the specification, eval_frequency is local to the package—it can be invoked only by other subprograms in the package, not from outside the package.

***Example 7–4 IF Statement that Determines Return Value of Function***

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date     EMPLOYEES.HIRE_DATE%TYPE;
  today      EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq  PLS_INTEGER;
BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  IF ((hire_date + (INTERVAL '120' MONTH)) < today) THEN
    eval_freq := 1;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for the syntax of the `IF` statement
- *Oracle Database PL/SQL Language Reference* for more information about using the `IF` statement

## Using the CASE Statement

The `CASE` statement chooses from a sequence of conditions, and executes the corresponding statement.

The simple `CASE` statement evaluates a single expression and compares it to several potential values. It has this syntax:

```
CASE expression
WHEN value THEN statement
[ WHEN value THEN statement ]...
[ ELSE statement [, statement ]... ]
END CASE;
```

The searched `CASE` statement evaluates multiple Boolean expressions and chooses the first one whose value is `TRUE`. For information about the searched `CASE` statement, see *Oracle Database PL/SQL Language Reference*.

> **Tip:** When you can use either a `CASE` statement or nested `IF` statements, use a `CASE` statement—it is both more readable and more efficient.

Suppose that, if an employee is evaluated only once a year, you want the `eval_frequency` function to suggest a salary increase, which depends on the `JOB_ID`.

Add the `CASE` statement shown in Example 7–5 to the `eval_frequency` function. (For information about the procedure that prints the strings, `DBMS_OUTPUT.PUT_LINE`, see *Oracle Database PL/SQL Packages and Types Reference*.)

***Example 7–5 CASE Statement that Determines Which String to Print***

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date     EMPLOYEES.HIRE_DATE%TYPE;
  today      EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq  PLS_INTEGER;
  j_id       EMPLOYEES.JOB_ID%TYPE;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  IF ((h_date + (INTERVAL '120' MONTH)) < today) THEN
    eval_freq := 1;

    SELECT JOB_ID INTO j_id
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;
```

```
    CASE j_id
       WHEN 'PU_CLERK' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 8% salary increase for employee # ' || emp_id);
       WHEN 'SH_CLERK' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 7% salary increase for employee # ' || emp_id);
       WHEN 'ST_CLERK' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 6% salary increase for employee # ' || emp_id);
       WHEN 'HR_REP' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 5% salary increase for employee # ' || emp_id);
       WHEN 'PR_REP' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 5% salary increase for employee # ' || emp_id);
       WHEN 'MK_REP' THEN DBMS_OUTPUT.PUT_LINE(
         'Consider 4% salary increase for employee # ' || emp_id);
       ELSE DBMS_OUTPUT.PUT_LINE(
         'Nothing to do for employee #' || emp_id);
    END CASE;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

**See Also:**

- "Using CASE Expressions in Queries" on page 4-29
- *Oracle Database PL/SQL Language Reference* for the syntax of the `CASE` statement
- *Oracle Database PL/SQL Language Reference* for more information about using the `CASE` statement

## Using the FOR LOOP Statement

The `FOR LOOP` statement repeats a sequence of statements once for each integer in the range *`lower_bound`* through *`upper_bound`*. Its syntax is:

```
FOR counter IN lower_bound..upper_bound LOOP
  statement [, statement ]...
END LOOP;
```

The statements between `LOOP` and `END LOOP` can use *`counter`*, but cannot change its value.

Suppose that, instead of only suggesting a salary increase, you want the `eval_frequency` function to report what the salary would be if it increased by the suggested amount every year for five years.

Change the `eval_frequency` function as shown in bold font in Example 7–6. (For information about the procedures that prints the strings, `DBMS_OUTPUT.PUT_LINE`, see *Oracle Database PL/SQL Packages and Types Reference*.)

***Example 7–6 FOR LOOP Statement that Computes Salary After Five Years***

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date      EMPLOYEES.HIRE_DATE%TYPE;
  today       EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq   PLS_INTEGER;
```

```
  j_id        EMPLOYEES.JOB_ID%TYPE;
  sal         EMPLOYEES.SALARY%TYPE;
  sal_raise   NUMBER(3,3) := 0;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  IF ((h_date + (INTERVAL '120' MONTH)) < today) THEN
    eval_freq := 1;

    SELECT JOB_ID INTO j_id
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;

    SELECT SALARY INTO sal
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;

    CASE j_id
      WHEN 'PU_CLERK' THEN sal_raise := 0.08;
      WHEN 'SH_CLERK' THEN sal_raise := 0.07;
      WHEN 'ST_CLERK' THEN sal_raise := 0.06;
      WHEN 'HR_REP'   THEN sal_raise := 0.05;
      WHEN 'PR_REP'   THEN sal_raise := 0.05;
      WHEN 'MK_REP'   THEN sal_raise := 0.04;
      ELSE NULL;
    END CASE;

    IF (sal_raise != 0) THEN
      BEGIN
        DBMS_OUTPUT.PUT_LINE('If salary ' || sal || ' increases by ' ||
          ROUND((sal_raise * 100),0) ||
          '% each year for 5 years, it will be:');

        FOR i IN 1..5 LOOP
          sal := sal * (1 + sal_raise);
          DBMS_OUTPUT.PUT_LINE(ROUND(sal, 2) || ' after ' || i || ' year(s)');
        END LOOP;
      END;
    END IF;

  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for the syntax of the `FOR LOOP` statement
- *Oracle Database PL/SQL Language Reference* for more information about using the `FOR LOOP` statement

## Using the WHILE LOOP Statement

The WHILE LOOP statement repeats a sequence of statements while a condition is TRUE. Its syntax is:

```
WHILE condition LOOP
  statement [, statement ]...
END LOOP;
```

> **Note:** If the statements between LOOP and END LOOP never cause *condition* to become FALSE, the WHILE LOOP statement runs indefinitely.

Suppose that the eval_frequency function uses the WHILE LOOP statement instead of the FOR LOOP statement, and ends after the proposed salary exceeds the maximum salary for the JOB_ID.

Change the eval_frequency function as shown in bold font in Example 7–7. (For information about the procedures that prints the strings, DBMS_OUTPUT.PUT_LINE, see *Oracle Database PL/SQL Packages and Types Reference*.)

***Example 7–7 WHILE LOOP Statement that Computes Salary to Maximum***

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date      EMPLOYEES.HIRE_DATE%TYPE;
  today       EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq   PLS_INTEGER;
  j_id        EMPLOYEES.JOB_ID%TYPE;
  sal         EMPLOYEES.SALARY%TYPE;
  sal_raise   NUMBER(3,3) := 0;
  sal_max     JOBS.MAX_SALARY%TYPE;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  IF ((h_date + (INTERVAL '120' MONTH)) < today) THEN
    eval_freq := 1;

    SELECT JOB_ID INTO j_id
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;

    SELECT SALARY INTO sal
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;

    SELECT MAX_SALARY INTO sal_max
    FROM JOBS
    WHERE JOB_ID = j_id;

    CASE j_id
      WHEN 'PU_CLERK' THEN sal_raise := 0.08;
      WHEN 'SH_CLERK' THEN sal_raise := 0.07;
```

```
      WHEN 'ST_CLERK' THEN sal_raise := 0.06;
      WHEN 'HR_REP'   THEN sal_raise := 0.05;
      WHEN 'PR_REP'   THEN sal_raise := 0.05;
      WHEN 'MK_REP'   THEN sal_raise := 0.04;
      ELSE NULL;
    END CASE;

    IF (sal_raise != 0) THEN
      BEGIN
        DBMS_OUTPUT.PUT_LINE('If salary ' || sal || ' increases by ' ||
          ROUND((sal_raise * 100),0) ||
          '% each year, it will be:');

        WHILE sal <= sal_max LOOP
          sal := sal * (1 + sal_raise);
          DBMS_OUTPUT.PUT_LINE(ROUND(sal, 2);
        END LOOP;

        DBMS_OUTPUT.PUT_LINE('Maximum salary for this job is ' || sal_max);
      END;
    END IF;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for the syntax of the `WHILE LOOP` statement
- *Oracle Database PL/SQL Language Reference* for more information about using the `WHILE LOOP` statement

## Using the Basic LOOP and EXIT WHEN Statements

The basic `LOOP` statement repeats a sequence of statements. Its syntax is:

```
LOOP
  statement [, statement ]...
END LOOP;
```

At least one *`statement`* must be an `EXIT` statement; otherwise, the `LOOP` statement runs indefinitely.

The `EXIT WHEN` statement (the `EXIT` statement with its optional `WHEN` clause) exits a loop when a condition is `TRUE` and transfers control to the end of the loop.

In the `eval_frequency` function, in the last iteration of the `WHILE LOOP` statement, the last computed value usually exceeds the maximum salary.

Change the `WHILE LOOP` statement to a basic `LOOP` statement that includes an `EXIT WHEN` statement, as shown in Example 7–8.

```
WHILE LOOP
```

***Example 7–8 Using the EXIT WHEN Statement***

```
FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
```

```
AS
  h_date      EMPLOYEES.HIRE_DATE%TYPE;
  today       EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq   PLS_INTEGER;
  j_id        EMPLOYEES.JOB_ID%TYPE;
  sal         EMPLOYEES.SALARY%TYPE;
  sal_raise   NUMBER(3,3) := 0;
  sal_max     JOBS.MAX_SALARY%TYPE;

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  IF ((h_date + (INTERVAL '120' MONTH)) < today) THEN
    eval_freq := 1;

    SELECT JOB_ID INTO j_id
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;

    SELECT SALARY INTO sal
    FROM EMPLOYEES
    WHERE EMPLOYEE_ID = emp_id;

    SELECT MAX_SALARY INTO sal_max
    FROM JOBS
    WHERE JOB_ID = j_id;

    CASE j_id
      WHEN 'PU_CLERK' THEN sal_raise := 0.08;
      WHEN 'SH_CLERK' THEN sal_raise := 0.07;
      WHEN 'ST_CLERK' THEN sal_raise := 0.06;
      WHEN 'HR_REP'   THEN sal_raise := 0.05;
      WHEN 'PR_REP'   THEN sal_raise := 0.05;
      WHEN 'MK_REP'   THEN sal_raise := 0.04;
      ELSE NULL;
    END CASE;

    IF (sal_raise != 0) THEN
      BEGIN
        DBMS_OUTPUT.PUT_LINE('If salary ' || sal || ' increases by ' ||
          ROUND((sal_raise * 100),0) ||
          '% each year, it will be:');

        LOOP
          sal := sal * (1 + sal_raise);
          EXIT WHEN sal > sal_max;
          DBMS_OUTPUT.PUT_LINE(ROUND(sal,2));
        END LOOP;

        DBMS_OUTPUT.PUT_LINE('Maximum salary for this job is ' || sal_max);
      END;
    END IF;
  ELSE
    eval_freq := 2;
  END IF;
```

```
  RETURN eval_freq;
END eval_frequency;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for the syntax of the `LOOP` statement
- *Oracle Database PL/SQL Language Reference* for the syntax of the `EXIT` statement
- *Oracle Database PL/SQL Language Reference* for more information about using the `LOOP` and `EXIT` statements

# Using Records and Cursors

Topics:



**See Also:** *Oracle Database PL/SQL Language Reference* for more information about records

## About Records

A **record** is a PL/SQL composite variable that can store data values of different types, similar to a `struct` type in C, C++, or Java. The internal components of a record are called **fields**. To access a record field, you use **dot notation**: *`record_name.field_name`*.

You can treat record fields like scalar variables. You can also pass entire records as subprogram parameters (if neither the sending nor receiving subprogram is a standalone stored subprogram).

Records are useful for holding data from table rows, or from certain columns of table rows. Each record field corresponds to a table column.

There are three ways to create a record:

- Declare a `RECORD` type, and then declare a variable of that type.

  The syntax is:

  ```
  TYPE record_name IS RECORD
    ( field_name data_type [:= initial_value]
   [, field_name data_type [:= initial_value ] ]... );

  variable_name record_name;
  ```

- Declare a variable of the type *table_name*%ROWTYPE.

  The fields of the record have the same names and data types as the columns of the table.

- Declare a variable of the type *cursor_name*%ROWTYPE.

  The fields of the record have the same names and data types as the columns of the table in the FROM clause of the cursor SELECT statement.

  **See Also:**

  - *Oracle Database PL/SQL Language Reference* for more information about defining RECORD types and declaring records of that type
  - *Oracle Database PL/SQL Language Reference* for the syntax of a RECORD type definition
  - *Oracle Database PL/SQL Language Reference* for more information about the %ROWTYPE attribute
  - *Oracle Database PL/SQL Language Reference* for the syntax of the %ROWTYPE attribute

## Tutorial: Declaring a RECORD Type

This tutorial shows how to use the SQL Developer tool Edit to declare a RECORD type, sal_info, whose fields can hold salary information for an employee—job ID, minimum and maximum salary for that job ID, current salary, and suggested raise.

**To declare RECORD type sal_info:**

1. On the Connections tab, expand the hr_conn information by clicking the plus sign (+) to the left of the hr_conn icon.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Packages**.

   A list of packages appears.

3. Right-click **EMP_EVAL**.

   A list of choices appears.

4. Click **Edit**.

   The EMP_EVAL pane opens, showing the CREATE PACKAGE statement that created the package:

```
CREATE OR REPLACE PACKAGE emp_eval AS

PROCEDURE eval_department(dept_id IN NUMBER);
FUNCTION calculate_score(evaluation_id IN NUMBER
                       , performance_id IN NUMBER)
                         RETURN NUMBER;

END emp_eval;
```

5. In the EMP_EVAL pane, immediately before END emp_eval, add this code:

```
TYPE sal_info IS RECORD
  ( j_id     jobs.job_id%type
  , sal_min  jobs.min_salary%type
  , sal_max  jobs.max_salary%type
```

```
  , sal      employees.salary%type
  , sal_raise NUMBER(3,3) );
```

A new EMP_EVAL pane opens, showing the changed CREATE PACKAGE statement. The title of the pane is in italic font, which indicates that the changes have not been saved to the database.

6. Click the icon **Compile**.

   The changed package specification compiles and is saved to the database. The title of the EMP_EVAL pane is no longer in italic font.

   Now you can declare records of the type sal_info, as in "Tutorial: Creating and Invoking a Subprogram with a Record Parameter" on page 7-32.

## Tutorial: Creating and Invoking a Subprogram with a Record Parameter

If you declared the RECORD type sal_info in "Tutorial: Declaring a RECORD Type" on page 7-31, this tutorial shows how to use the SQL Developer tool Edit to do the following:

- Create a procedure, salary_schedule, which has a parameter of type sal_info.
- Change the eval_frequency function so that it declares a record, emp_sal, of the type sal_info, populates its fields, and passes it to the salary_schedule procedure.

Because eval_frequency will invoke salary_schedule, the declaration of salary_schedule must precede the declaration of eval_frequency (otherwise, the package will not compile). However, the definition of salary_schedule can be anywhere in the package body.

### To create salary_schedule and change eval_frequency:

1. On the Connections tab, expand the hr_conn information by clicking the plus sign (+) to the left of the hr_conn icon.

   Under the hr_conn icon, a list of schema object types appears.

2. Expand **Packages**.

   A list of packages appears.

3. Expand **EMP_EVAL**.

   A list appears.

4. Right-click **EMP_EVAL Body**.

   A list of choices appears.

5. Click **Edit**.

   The EMP_EVAL Body pane appears, showing the code for the package body.

6. In the EMP_EVAL Body pane, immediately before END emp_eval, add this definition of the salary_schedule procedure:

```
PROCEDURE salary_schedule (emp IN sal_info) AS
  accumulating_sal  NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('If salary ' || emp.sal ||
    ' increases by ' || ROUND((emp.sal_raise * 100),0) ||
    '% each year, it will be:');
```

```
  accumulating_sal := emp.sal;

  WHILE accumulating_sal <= emp.sal_max LOOP
    accumulating_sal := accumulating_sal * (1 + emp.sal_raise);
    DBMS_OUTPUT.PUT_LINE(ROUND(accumulating_sal,2) ||', ');
  END LOOP;
END salary_schedule;
```

A new EMP_EVAL Body pane opens, showing the changed CREATE PACKAGE BODY statement. The title of the pane is in italic font, which indicates that the changes have not been saved to the database.

7. In the EMP_EVAL Body pane, enter the code shown in bold font, in this position:

```
create or replace
PACKAGE BODY EMP_EVAL AS

FUNCTION eval_frequency (emp_id EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER;
PROCEDURE salary_schedule(emp IN sal_info);
PROCEDURE add_eval(employee_id IN NUMBER, today IN DATE);

PROCEDURE eval_department (dept_id IN NUMBER) AS
```

8. Edit the eval_frequency function, making the changes shown in bold font:

```
FUNCTION eval_frequency (emp_id EMPLOYEES.EMPLOYEE_ID%TYPE)
  RETURN PLS_INTEGER
AS
  h_date     EMPLOYEES.HIRE_DATE%TYPE;
  today      EMPLOYEES.HIRE_DATE%TYPE;
  eval_freq  PLS_INTEGER;
  emp_sal    SAL_INFO;  -- replaces sal, sal_raise, and sal_max

BEGIN
  SELECT SYSDATE INTO today FROM DUAL;

  SELECT HIRE_DATE INTO h_date
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;

  IF ((h_date + (INTERVAL '120' MONTH)) < today) THEN
     eval_freq := 1;

     /* populate emp_sal */

     SELECT JOB_ID INTO emp_sal.j_id FROM EMPLOYEES
     WHERE EMPLOYEE_ID = emp_id;

     SELECT MIN_SALARY INTO emp_sal.sal_min FROM JOBS
     WHERE JOB_ID = emp_sal.j_id;

     SELECT MAX_SALARY INTO emp_sal.sal_max FROM JOBS
     WHERE JOB_ID = emp_sal.j_id;

     SELECT SALARY INTO emp_sal.sal FROM EMPLOYEES
     WHERE EMPLOYEE_ID = emp_id;

     emp_sal.sal_raise := 0;  -- default
```

```
    CASE emp_sal.j_id
      WHEN 'PU_CLERK' THEN emp_sal.sal_raise := 0.08;
      WHEN 'SH_CLERK' THEN emp_sal.sal_raise := 0.07;
      WHEN 'ST_CLERK' THEN emp_sal.sal_raise := 0.06;
      WHEN 'HR_REP' THEN emp_sal.sal_raise := 0.05;
      WHEN 'PR_REP' THEN emp_sal.sal_raise := 0.05;
      WHEN 'MK_REP' THEN emp_sal.sal_raise := 0.04;
      ELSE NULL;
    END CASE;

    IF (emp_sal.sal_raise != 0) THEN
      salary_schedule(emp_sal);
    END IF;
  ELSE
    eval_freq := 2;
  END IF;

  RETURN eval_freq;
END eval_frequency;
```

9. Click **Compile**.

## About Cursors

When Oracle Database executes a SQL statement, it stores the result set and processing information in an unnamed private SQL area. A pointer to this unnamed area, called a **cursor**, lets you retrieve the rows of the result set one at a time. **Cursor attributes** return information about the state of the cursor.

Every time you run either a SQL DML statement or a PL/SQL `SELECT INTO` statement, PL/SQL opens an **implicit cursor**. You can get information about this cursor from its attributes, but you cannot control it. After the statement runs, the database closes the cursor; however, its attribute values remain available until another DML or `SELECT INTO` statement runs.

PL/SQL also lets you declare explicit cursors. An **explicit cursor** has a name and is associated with a query (SQL `SELECT` statement)—usually one that returns multiple rows. After declaring an explicit cursor, you must open it (with the `OPEN` statement), fetch rows one at a time from the result set (with the `FETCH` statement), and close the cursor (with the `CLOSE` statement). After closing the cursor, you can neither fetch records from the result set nor see the cursor attribute values.

The syntax for the value of an implicit cursor attribute is `SQL`*`attribute`* (for example, `SQL%FOUND`). `SQL`*`attribute`* always refers to the most recently run DML or `SELECT INTO` statement.

The syntax for the value of an explicit cursor attribute is *`cursor_name`* immediately followed by *`attribute`* (for example, `c1%FOUND`).

Table 7–1 lists the cursor attributes and the values that they can return. (Implicit cursors have additional attributes that are beyond the scope of this book.)

*Table 7–1 Cursor Attribute Values*

| Attribute | Values for Explicit Cursor | Values for Implicit Cursor |
|---|---|---|
| %FOUND | If cursor is not open, INVALID_CURSOR.<br>If cursor is open but no fetch was attempted, NULL.<br>If the most recent fetch returned a row, TRUE.<br>If the most recent fetch did not return a row, FALSE. | If no DML or SELECT INTO statement has run, NULL.<br>If the most recent DML or SELECT INTO statement returned a row, TRUE.<br>If the most recent DML or SELECT INTO statement did not return a row, FALSE. |
| %NOTFOUND | If cursor is not open, INVALID_CURSOR.<br>If cursor is open but no fetch was attempted, NULL.<br>If the most recent fetch returned a row, FALSE.<br>If the most recent fetch did not return a row, TRUE. | If no DML or SELECT INTO statement has run, NULL.<br>If the most recent DML or SELECT INTO statement returned a row, FALSE.<br>If the most recent DML or SELECT INTO statement did not return a row, TRUE. |
| %ROWCOUNT | If cursor is not open, INVALID_CURSOR; otherwise, a number greater than or equal to zero. | NULL if no DML or SELECT INTO statement has run; otherwise, a number greater than or equal to zero. |
| %ISOPEN | If cursor is open, TRUE; if not, FALSE. | Always FALSE. |

**See Also:**

- "About Queries" on page 4-1
- "About Data Manipulation Language (DML) Statements" on page 5-1
- *Oracle Database PL/SQL Language Reference* for more information about the SELECT INTO statement
- *Oracle Database Advanced Application Developer's Guide* for more information about using cursors in applications
- *Oracle Database PL/SQL Language Reference* for more information about managing cursors in PL/SQL

## Using an Explicit Cursor to Retrieve Result Set Rows One at a Time

The following procedure uses each necessary statement in its simplest form, but provides references to its complete syntax.

**To use an explicit cursor to retrieve result set rows one at a time:**

1. In the declarative part:
   a. Declare the cursor:

      ```
      CURSOR cursor_name IS query;
      ```

      For complete explicit cursor declaration syntax, see *Oracle Database PL/SQL Language Reference*.

   b. Declare a record to hold the row returned by the cursor:

```
record_name cursor_name%ROWTYPE;
```

For complete `%ROWTYPE` syntax, see *Oracle Database PL/SQL Language Reference*.

2. In the executable part:

   a. Open the cursor:

   ```
   OPEN cursor_name;
   ```

   For complete `OPEN` statement syntax, see *Oracle Database PL/SQL Language Reference*.

   b. Fetch rows from the cursor (rows from the result set) one at a time, using a `LOOP` statement that has syntax similar to this:

   ```
   LOOP
     FETCH cursor_name INTO record_name;
     EXIT WHEN cursor_name%NOTFOUND;
     -- Process row that is in record_name:
     statement;
     [ statement; ]...
   END LOOP;
   ```

   For complete `FETCH` statement syntax, see *Oracle Database PL/SQL Language Reference*.

   c. Close the cursor:

   ```
   CLOSE cursor_name;
   ```

   For complete `CLOSE` statement syntax, see *Oracle Database PL/SQL Language Reference*.

## Tutorial: Using an Explicit Cursor to Retrieve Result Set Rows One at a Time

This tutorial shows how to implement the procedure `EMP_EVAL.eval_department`, which uses an explicit cursor, `emp_cursor`.

### To implement the EMP_EVAL.eval_department procedure:

1. In the `EMP_EVAL` package specification, change the declaration of the `eval_department` procedure as shown in bold font:

   ```
   PROCEDURE eval_department(dept_id IN employees.department_id%TYPE);
   ```

   (For instructions for changing a package specification, see "Tutorial: Changing a Package Specification" on page 7-13.)

2. In the `EMP_EVAL` package body, change the definition of the `eval_department` procedure as shown in bold font:

   ```
   PROCEDURE eval_department (dept_id IN employees.department_id%TYPE)
   AS
     CURSOR emp_cursor IS
       SELECT * FROM EMPLOYEES
       WHERE DEPARTMENT_ID = dept_id;

     emp_record  EMPLOYEES%ROWTYPE;  -- for row returned by cursor
     all_evals   BOOLEAN;  -- true if all employees in dept need evaluations
     today       DATE;
   ```

```
BEGIN
  today := SYSDATE;

  IF (EXTRACT(MONTH FROM today) < 6) THEN
    all_evals := FALSE; -- only new employees need evaluations
  ELSE
    all_evals := TRUE;  -- all employees need evaluations
  END IF;

  OPEN emp_cursor;

  DBMS_OUTPUT.PUT_LINE (
    'Determining evaluations necessary in department # ' ||
    dept_id );

  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;

    IF all_evals THEN
      add_eval(emp_record.employee_id, today);
    ELSIF (eval_frequency(emp_record.employee_id) = 2) THEN
      add_eval(emp_record.employee_id, today);
    END IF;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Processed ' || emp_cursor%ROWCOUNT || ' records.');

  CLOSE emp_cursor;
END eval_department;
```

(For a step-by-step example of changing a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16.)

3. Compile the EMP_EVAL package specification.
4. Compile the EMP_EVAL package body.

## About Cursor Variables

A **cursor variable** is like a cursor (see "About Cursors" on page 7-34), except that it is not limited to one query. You can open a cursor variable for a query, process the result set, and then use the cursor variable for another query. Cursor variables are useful for passing query results between subprograms.

To declare a cursor variable, you declare a REF CURSOR type, and then declare a variable of that type (therefore, a cursor variable is often called a REF CURSOR). A REF CURSOR type can be either strong or weak.

A **strong REF CURSOR type** specifies a **return type**, which is the RECORD type of its cursor variables. The PL/SQL compiler does not allow you to use these **strongly typed** cursor variables for queries that return rows that are not of the return type. Strong REF CURSOR types are less error-prone than weak ones, but weak ones are more flexible.

A **weak REF CURSOR type** does not specify a return type. The PL/SQL compiler accepts **weakly typed** cursor variables in any queries. Weak REF CURSOR types are interchangeable; therefore, instead of creating weak REF CURSOR types, you can use the predefined type weak cursor type SYS_REFCURSOR.

After declaring a cursor variable, you must open it for a specific query (with the `OPEN FOR` statement), fetch rows one at a time from the result set (with the `FETCH` statement), and then either close the cursor (with the `CLOSE` statement) or open it for another specific query (with the `OPEN FOR` statement). Opening the cursor variable for another query closes it for the previous query. After closing a cursor variable for a specific query, you can neither fetch records from the result set of that query nor see the cursor attribute values for that query.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about using cursor variables
- *Oracle Database PL/SQL Language Reference* for the syntax of cursor variable declaration

## Using a Cursor Variable to Retrieve Result Set Rows One at a Time

The following procedure uses each of the necessary statements in its simplest form, but provides references to their complete syntax.

**To use a cursor variable to retrieve result set rows one at a time:**

1. In the declarative part:
   a. Declare the `REF CURSOR` type:

      ```
      TYPE cursor_type IS REF CURSOR [ RETURN return_type ];
      ```

      For complete `REF CURSOR` type declaration syntax, see *Oracle Database PL/SQL Language Reference*.

   b. Declare a cursor variable of that type:

      ```
      cursor_variable cursor_type;
      ```

      For complete cursor variable declaration syntax, see *Oracle Database PL/SQL Language Reference*.

   c. Declare a record to hold the row returned by the cursor:

      ```
      record_name return_type;
      ```

      For complete information about record declaration syntax, see *Oracle Database PL/SQL Language Reference*.

2. In the executable part:
   a. Open the cursor variable for a specific query:

      ```
      OPEN cursor_variable FOR query;
      ```

      For complete information about `OPEN FOR` statement syntax, see *Oracle Database PL/SQL Language Reference*.

   b. Fetch rows from the cursor variable (rows from the result set) one at a time, using a `LOOP` statement that has syntax similar to this:

      ```
      LOOP
        FETCH cursor_variable INTO record_name;
        EXIT WHEN cursor_variable%NOTFOUND;
        -- Process row that is in record_name:
        statement;
        [ statement; ]...
      ```

```
END LOOP;
```

For complete information about `FETCH` statement syntax, see *Oracle Database PL/SQL Language Reference*.

**c.** Close the cursor variable:

```
CLOSE cursor_variable;
```

Alternatively, you can open the cursor variable for another query, which closes it for the current query.

For complete information about `CLOSE` statement syntax, see *Oracle Database PL/SQL Language Reference*.

## Tutorial: Using a Cursor Variable to Retrieve Result Set Rows One at a Time

This tutorial shows how to change the `EMP_EVAL.eval_department` procedure so that it uses a cursor variable instead of an explicit cursor, which lets it process multiple departments. The change includes adding a procedure that uses the cursor variable.

This tutorial also shows how to make `EMP_EVAL.eval_department` and `EMP_EVAL.add_eval` more efficient: Instead of passing one field of a record to `add_eval` and having `add_eval` use three queries to extract three other fields of the same record, `eval_department` passes the entire record to `add_eval`, and `add_eval` uses dot notation to access the values of the other three fields.

### To change the EMP_EVAL.eval_department procedure to use a cursor variable:

**1.** In the `EMP_EVAL` package specification, add the procedure declaration and the `REF CURSOR` type definition, as shown in bold font:

```
create or replace
PACKAGE emp_eval AS

  PROCEDURE eval_department (dept_id IN employees.department_id%TYPE);

  PROCEDURE eval_everyone;

  FUNCTION calculate_score(eval_id IN scores.evaluation_id%TYPE
                         , perf_id IN scores.performance_id%TYPE)
                           RETURN NUMBER;
  TYPE SAL_INFO IS RECORD
      ( j_id jobs.job_id%type
      , sal_min jobs.min_salary%type
      , sal_max jobs.max_salary%type
      , salary employees.salary%type
      , sal_raise NUMBER(3,3));


  TYPE emp_refcursor_type IS REF CURSOR RETURN employees%ROWTYPE;
END emp_eval;
```

(For instructions for changing a package specification, see "Tutorial: Changing a Package Specification" on page 7-13.)

**2.** In the `EMP_EVAL` package body, add a forward declaration for the procedure `eval_loop_control` and change the declaration of the procedure `add_eval`, as shown in bold font:

```
create or replace
PACKAGE BODY EMP_EVAL AS
```

```
  FUNCTION eval_frequency (emp_id IN EMPLOYEES.EMPLOYEE_ID%TYPE)
    RETURN PLS_INTEGER;

  PROCEDURE salary_schedule(emp IN sal_info);

  PROCEDURE add_eval(emp_record IN EMPLOYEES%ROWTYPE, today IN DATE);

  PROCEDURE eval_loop_control(emp_cursor IN emp_refcursor_type);
...
```

(For a step-by-step example of changing a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16.)

3. Change the `eval_department` procedure to retrieve three separate result sets based on the department, and to call the `eval_loop_control` procedure, as shown in bold font:

```
PROCEDURE eval_department(dept_id IN employees.department_id%TYPE) AS
  emp_cursor    emp_refcursor_type;
  current_dept  departments.department_id%TYPE;

BEGIN
  current_dept := dept_id;

  FOR loop_c IN 1..3 LOOP
    OPEN emp_cursor FOR
      SELECT *
      FROM employees
      WHERE current_dept = dept_id;

    DBMS_OUTPUT.PUT_LINE
      ('Determining necessary evaluations in department #' ||
       current_dept);

    eval_loop_control(emp_cursor);

    DBMS_OUTPUT.PUT_LINE
      ('Processed ' || emp_cursor%ROWCOUNT || ' records.');

    CLOSE emp_cursor;
    current_dept := current_dept + 10;
  END LOOP;
END eval_department;
```

4. Change the `add_eval` as shown in bold font:

```
PROCEDURE add_eval(emp_record IN employees%ROWTYPE, today IN DATE)
AS
-- (Delete local variables)
BEGIN
  INSERT INTO EVALUATIONS (
    evaluation_id,
    employee_id,
    evaluation_date,
    job_id,
    manager_id,
    department_id,
    total_score
  )
  VALUES (
    evaluations_seq.NEXTVAL,   -- evaluation_id
```

```
    emp_record.employee_id,    -- employee_id
    today,                     -- evaluation_date
    emp_record.job_id,         -- job_id
    emp_record.manager_id,     -- manager_id
    emp_record.department_id,  -- department_id
    0                           -- total_score
);
END add_eval;
```

5. Before `END EMP_EVAL`, add the following procedure, which fetches the individual records from the result set and processes them:

```
PROCEDURE eval_loop_control (emp_cursor IN emp_refcursor_type) AS
   emp_record      EMPLOYEES%ROWTYPE;
   all_evals       BOOLEAN;
   today           DATE;
BEGIN
  today := SYSDATE;

  IF (EXTRACT(MONTH FROM today) < 6) THEN
    all_evals := FALSE;
  ELSE
    all_evals := TRUE;
  END IF;

  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;

    IF all_evals THEN
      add_eval(emp_record, today);
    ELSIF (eval_frequency(emp_record.employee_id) = 2) THEN
      add_eval(emp_record, today);
    END IF;
  END LOOP;
END eval_loop_control;
```

6. Before `END EMP_EVAL`, add the following procedure, which retrieves a result set that contains all employees in the company:

```
PROCEDURE eval_everyone AS
  emp_cursor emp_refcursor_type;
BEGIN
  OPEN emp_cursor FOR SELECT * FROM employees;
  DBMS_OUTPUT.PUT_LINE('Determining number of necessary evaluations.');
  eval_loop_control(emp_cursor);
  DBMS_OUTPUT.PUT_LINE('Processed ' || emp_cursor%ROWCOUNT || ' records.');
  CLOSE emp_cursor;
END eval_everyone;
```

7. Compile the `EMP_EVAL` package specification.
8. Compile the `EMP_EVAL` package body.

## Using Associative Arrays

An associative array is a type of collection.

Topics:

- About Collections

- About Associative Arrays
- Declaring Associative Arrays
- Populating Associative Arrays
- Traversing Dense Associative Arrays
- Traversing Sparse Associative Arrays

**See Also:** For more information about collections:

- *Oracle Database Concepts*
- *Oracle Database PL/SQL Language Reference*

## About Collections

A **collection** is a PL/SQL composite variable that stores elements of the same type in a specified order, similar to a one-dimensional array. The internal components of a collection are called **elements**. Each element has a unique subscript that identifies its position in the collection. To access a collection element, you use **subscript notation**: *`collection_name`*(*`element_subscript`*).

You can treat collection elements like scalar variables. You can also pass entire collections as subprogram parameters (if neither the sending nor receiving subprogram is a standalone stored subprogram).

A **collection method** is a built-in PL/SQL subprogram that either returns information about a collection or operates on a collection. To invoke a collection method, you use **dot notation**: *`collection_name.method_name`*. For example, *`collection_name`*.`COUNT` returns the number of elements in the collection.

PL/SQL has three types of collections:

- Associative arrays (formerly called "PL/SQL tables" or "index-by tables")
- Nested tables
- Variable arrays (varrays)

This document explains only associative arrays.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL collection types
- *Oracle Database PL/SQL Language Reference* for more information about collection methods

## About Associative Arrays

An **associative array** is an unbounded set of key-value pairs. Each key is unique, and serves as the subscript of the element that holds the corresponding value. Therefore, you can access elements without knowing their positions in the array, and without traversing the array.

The data type of the key can be either `PLS_INTEGER` or `VARCHAR2 (`*`length`*`)`.

If the key type is `PLS_INTEGER`, the associative array is **indexed by integer**, and it is **dense**; that is, it has no gaps between elements—every element between the first and last element is defined and has a value (which can be `NULL`).

If the key type is VARCHAR2 (*length*), the associative array is **indexed by string** (of *length* characters), and it is **sparse**; that is, it might have gaps between elements.

When traversing a dense associative array, you do not need to beware of gaps between elements; when traversing a sparse associative array, you do.

To assign a value to an associative array element, you can use an assignment operator:

```
array_name(key) := value
```

If *key* is not in the array, the assignment statement adds the *key*-*value* pair to the array. Otherwise, the statement changes the value of *array_name*(*key*) to *value*.

Associative arrays are useful for storing data temporarily. They do not use the disk space or network operations that tables require. However, because associative arrays are intended for temporary storage, you cannot manipulate them with DML statements or use SELECT INTO statements to assign their values to variables.

If you declare an associative array in a package, and assign values to the variable in the package body, then the associative array exists for the life of the database session. Otherwise, it exists for the life of the subprogram in which you declare it.

> **See Also:** *Oracle Database PL/SQL Language Reference* for more information about associative arrays

## Declaring Associative Arrays

To declare an associative array, you declare an associative array type, and then declare a variable of that type. The simplest syntax is:

```
TYPE array_type IS TABLE OF element_type INDEX BY key_type;

array_name  array_type;
```

An efficient way to declare an associative array is with a cursor, using the following procedure. The procedure uses each necessary statement in its simplest form, but provides references to its complete syntax.

### To use a cursor to declare an associative array:

1. In the declarative part:
   a. Declare the cursor:

      ```
      CURSOR cursor_name IS query;
      ```

      For complete explicit cursor declaration syntax, see *Oracle Database PL/SQL Language Reference*.

   b. Declare the associative array type:

      ```
      TYPE array_type IS TABLE OF cursor_name%ROWTYPE
        INDEX BY { PLS_INTEGER | VARCHAR2 length }
      ```

      For complete associative array type declaration syntax, see *Oracle Database PL/SQL Language Reference*.

   c. Declare an associative array variable of that type:

      ```
      array_name  array_type;
      ```

      For complete variable declaration syntax, see *Oracle Database PL/SQL Language Reference*.

Example 7–9 uses the preceding procedure to declare two associative arrays, `employees_jobs` and `jobs_`, and then declares a third associative array, `job_titles_type`, without using a cursor. The first two arrays are indexed by integer; the third is indexed by string.

---

**Note:** The `ORDER BY` clause in the declaration of `employees_jobs_cursor` determines the storage order of the elements of the associative array `employee_jobs`.

---

***Example 7–9 Declaring Associative Arrays***

```
DECLARE
  -- Declare cursor:

  CURSOR employees_jobs_cursor IS
    SELECT FIRST_NAME, LAST_NAME, JOB_ID
    FROM EMPLOYEES
    ORDER BY JOB_ID, LAST_NAME, FIRST_NAME;

  -- Declare associative array type:

  TYPE employees_jobs_type IS TABLE OF employees_jobs_cursor%ROWTYPE
    INDEX BY PLS_INTEGER;

  -- Declare associative array:

  employees_jobs  employees_jobs_type;

  -- Use same procedure to declare another associative array:

  CURSOR jobs_cursor IS
    SELECT JOB_ID, JOB_TITLE
    FROM JOBS;

  TYPE jobs_type IS TABLE OF jobs_cursor%ROWTYPE
    INDEX BY PLS_INTEGER;

  jobs_  jobs_type;

-- Declare associative array without using cursor:

  TYPE job_titles_type IS TABLE OF JOBS.JOB_TITLE%TYPE
    INDEX BY JOBS.JOB_ID%TYPE;  -- jobs.job_id%type is varchar2(10)

  job_titles  job_titles_type;

BEGIN
  NULL;
END;
/
```

**See Also:**

- "About Cursors" on page 7-34
- *Oracle Database PL/SQL Language Reference* for associative array declaration syntax

## Populating Associative Arrays

The most efficient way to populate a dense associative array is with a cursor and the `FETCH` statement with the `BULK COLLECT INTO` clause, using the following procedure. The procedure uses each necessary statement in its simplest form, but provides references to its complete syntax.

You cannot use the following procedure to populate a sparse associative array. Instead, you must use an assignment statement inside a loop statement. For information about loop statements, see "Controlling Program Flow" on page 7-22.

**To use a cursor to populate an associative array indexed by integer:**

1. If you have not done so, declare an associative array with a cursor, using the procedure in "Declaring Associative Arrays" on page 7-43.
2. In the executable part of the PL/SQL unit in which you declared the associative array:
   a. Open the cursor:

      ```
      OPEN cursor_name;
      ```

      For complete `OPEN` statement syntax, see *Oracle Database PL/SQL Language Reference*.

   b. Fetch all rows from the cursor into the associative array variable at once, using a `FETCH` statement with the `BULK COLLECT INTO` clause:

      ```
      FETCH cursor_name BULK COLLECT INTO aa_variable;
      ```

      For complete `FETCH` statement syntax, see *Oracle Database PL/SQL Language Reference*.

   c. Close the cursor:

      ```
      CLOSE cursor_name;
      ```

      For complete `CLOSE` statement syntax, see *Oracle Database PL/SQL Language Reference*.

Example 7–10 uses the preceding procedure to populate the associative arrays `employees_jobs` and `jobs_`, which are indexed by integer. Then it uses an assignment statement inside a `FOR LOOP` statement to populate the associative array `job_titles_type`, which is indexed by string.

FOR LOOP

***Example 7–10 Populating Associative Arrays***

```
-- Declarative part from Example 7-9 goes here.
BEGIN
  -- Populate associative arrays indexed by integer:

  OPEN employees_jobs_cursor;
  FETCH employees_jobs_cursor BULK COLLECT INTO employees_jobs;
  CLOSE employees_jobs_cursor;

  OPEN jobs_cursor;
  FETCH jobs_cursor BULK COLLECT INTO jobs_;
  CLOSE jobs_cursor;

  -- Populate associative array indexed by string:
```

```
  FOR i IN 1..jobs_.COUNT() LOOP
    job_titles(jobs_(i).job_id) := jobs_(i).job_title;
  END LOOP;
END;
/
```

**See Also:** "About Cursors" on page 7-34

## Traversing Dense Associative Arrays

A **dense associative array** (indexed by integer) has no gaps between elements—every element between the first and last element is defined and has a value (which can be `NULL`). You can traverse a dense array with a `FOR LOOP` statement, as in Example 7–11.

When inserted in the executable part of Example 7–10, after the code that populates the `employees_jobs` array, the `FOR LOOP` statement in Example 7–11 prints the elements of the `employees_jobs` array in the order in which they were stored. Their storage order was determined by the `ORDER BY` clause in the declaration of `employees_jobs_cursor`, which was used to declare `employees_jobs` (see Example 7–9).

`FOR LOOP`

`FOR LOOP`

The upper bound of the `FOR LOOP` statement, `employees_jobs.COUNT`, invokes a collection method that returns the number of elements in the array. For more information about `COUNT`, see *Oracle Database PL/SQL Language Reference*.

***Example 7–11 Traversing a Dense Associative Array***

```
-- Code that populates employees_jobs must precede this code:

FOR i IN 1..employees_jobs.COUNT LOOP
  DBMS_OUTPUT.PUT_LINE(
    RPAD(employees_jobs(i).first_name, 23) ||
    RPAD(employees_jobs(i).last_name,  28) ||
     employees_jobs(i).job_id);
  END LOOP;
```

Result:

```
William                 Gietz                       AC_ACCOUNT
Shelley                 Higgins                     AC_MGR
Jennifer                Whalen                      AD_ASST
Steven                  King                        AD_PRES
Lex                     De Haan                     AD_VP
Neena                   Kochhar                     AD_VP
John                    Chen                        FI_ACCOUNT
...
Jose Manuel             Urman                       FI_ACCOUNT
Nancy                   Greenberg                   FI_MGR
Susan                   Mavris                      HR_REP
David                   Austin                      IT_PROG
...
Valli                   Pataballa                   IT_PROG
Michael                 Hartstein                   MK_MAN
Pat                     Fay                         MK_REP
Hermann                 Baer                        PR_REP
Shelli                  Baida                       PU_CLERK
```

```
...
Sigal                Tobias               PU_CLERK
Den                  Raphaely             PU_MAN
Gerald               Cambrault            SA_MAN
...
Eleni                Zlotkey              SA_MAN
Ellen                Abel                 SA_REP
...
Clara                Vishney              SA_REP
Sarah                Bell                 SH_CLERK
...
Peter                Vargas               ST_CLERK
Adam                 Fripp                ST_MAN
...
Matthew              Weiss                ST_MAN
```

## Traversing Sparse Associative Arrays

A **sparse associative array** (indexed by string) might have gaps between elements. You can traverse it with a `WHILE LOOP` statement, as in Example 7–12.

To run the code in Example 7–12, which prints the elements of the `job_titles` array:

1. At the end of the declarative part of Example 7–9, insert this variable declaration:

   ```
   i jobs_.job_id%TYPE;
   ```

2. In the executable part of Example 7–10, after the code that populates the `job_titles` array, insert the code from Example 7–12.

Example 7–12 includes two collection method invocations, `job_titles.FIRST` and `job_titles.NEXT(i)`. `job_titles.FIRST` returns the first element of `job_titles`, and `job_titles.NEXT(i)` returns the subscript that succeeds `i`. For more information about `FIRST`, see *Oracle Database PL/SQL Language Reference*. For more information about `NEXT`, see *Oracle Database PL/SQL Language Reference*.

```
WHILE LOOP
```

***Example 7–12 Traversing a Sparse Associative Array***

```
/* Declare this variable in declarative part:

   i jobs_.job_id%TYPE;

   Add this code to the executable part,
   after code that populates job_titles:
*/

i := job_titles.FIRST;

WHILE i IS NOT NULL LOOP
  DBMS_OUTPUT.PUT_LINE(RPAD(i, 12) || job_titles(i));
  i := job_titles.NEXT(i);
END LOOP;
```

Result:

```
AC_ACCOUNT  Public Accountant
AC_MGR      Accounting Manager
AD_ASST     Administration Assistant
AD_PRES     President
AD_VP       Administration Vice President
```

```
FI_ACCOUNT  Accountant
FI_MGR      Finance Manager
HR_REP      Human Resources Representative
IT_PROG     Programmer
MK_MAN      Marketing Manager
MK_REP      Marketing Representative
PR_REP      Public Relations Representative
PU_CLERK    Purchasing Clerk
PU_MAN      Purchasing Manager
SA_MAN      Sales Manager
SA_REP      Sales Representative
SH_CLERK    Shipping Clerk
ST_CLERK    Stock Clerk
ST_MAN      Stock Manager
```

# Handling Exceptions (Run-Time Errors)

Topics:

- About Exceptions and Exception Handlers
- Handling Predefined Exceptions
- Declaring and Handling User-Defined Exceptions

> **See Also:** *Oracle Database PL/SQL Language Reference* for more information about handling PL/SQL errors

## About Exceptions and Exception Handlers

When a run-time error occurs in PL/SQL code, an **exception** is raised. If the subprogram (or block) in which the exception is raised has an exception-handling part, control transfers to it; otherwise, execution stops.

Run-time errors can arise from design faults, coding mistakes, hardware failures, and many other sources. Because you cannot anticipate all possible errors, Oracle recommends including exception-handling parts in your subprograms ("About Subprogram Structure" on page 7-5 shows where to put the exception-handling part).

Oracle Database has many **predefined exceptions**, which it raises automatically when a program violates database rules or exceeds system-dependent limits. For example, if a `SELECT INTO` statement returns no rows, Oracle Database raises the predefined exception `NO_DATA_FOUND`. For a summary of predefined PL/SQL exceptions, see *Oracle Database PL/SQL Language Reference*.

PL/SQL lets you define (declare) your own exceptions. An exception declaration has this syntax:

```
exception_name EXCEPTION;
```

Unlike a predefined exception, a **user-defined exception** must be raised explicitly, using either the `RAISE` statement or the `DBMS_STANDARD.RAISE_APPLICATION_ERROR` procedure. For example:

```
IF condition THEN RAISE exception_name;
```

For information about the `DBMS_STANDARD.RAISE_APPLICATION_ERROR` procedure, see *Oracle Database PL/SQL Language Reference*.

The exception-handling part of a subprogram contains one or more exception handlers. An **exception handler** has this syntax:

```
WHEN { exception_name [ OR exception_name ]... | OTHERS } THEN
  statement; [ statement; ]...
```

A WHEN OTHERS exception handler handles unexpected run-time errors. If used, it must be last. For example:

```
EXCEPTION
  WHEN exception_1 THEN
    statement; [ statement; ]...
  WHEN exception_2 OR exception_3 THEN
    statement; [ statement; ]...
  WHEN OTHERS THEN
    statement; [ statement; ]...
END;
```

An alternative to the WHEN OTHERS exception handler is the EXCEPTION_INIT pragma, which associates a user-defined exception name with an Oracle Database error number.

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about exception declaration syntax
- *Oracle Database PL/SQL Language Reference* for more information about exception handler syntax
- *Oracle Database PL/SQL Language Reference* for more information about the EXCEPTION_INIT pragma

## Handling Predefined Exceptions

Example 7–13 shows, in bold font, how to change the EMP_EVAL.eval_department procedure to handle the predefined exception NO_DATA_FOUND. Make this change and compile the changed procedure. (For an example of how to change a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16.)

***Example 7–13 Handling Predefined Exception NO_DATA_FOUND***

```
PROCEDURE eval_department(dept_id IN employees.department_id%TYPE) AS
  emp_cursor    emp_refcursor_type;
  current_dept  departments.department_id%TYPE;

BEGIN
  current_dept := dept_id;

  FOR loop_c IN 1..3 LOOP
    OPEN emp_cursor FOR
      SELECT *
      FROM employees
      WHERE current_dept = dept_id;

    DBMS_OUTPUT.PUT_LINE
      ('Determining necessary evaluations in department #' ||
       current_dept);

    eval_loop_control(emp_cursor);

    DBMS_OUTPUT.PUT_LINE
      ('Processed ' || emp_cursor%ROWCOUNT || ' records.');
```

```
    CLOSE emp_cursor;
    current_dept := current_dept + 10;
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('The query did not return a result set');
END eval_department;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about predefined exceptions
- *Oracle Database PL/SQL Language Reference* for more information about handling raised exceptions

## Declaring and Handling User-Defined Exceptions

Example 7–14 shows, in bold font, how to change the `EMP_EVAL.calculate_score` function to declare and handle two user-defined exceptions, `wrong_weight` and `wrong_score`. Make this change and compile the changed function. (For an example of how to change a package body, see "Tutorial: Declaring Variables and Constants in a Subprogram" on page 7-16.)

***Example 7–14 Handling User-Defined Exceptions***

```
FUNCTION calculate_score ( evaluation_id IN scores.evaluation_id%TYPE
                         , performance_id IN scores.performance_id%TYPE )
                         RETURN NUMBER AS

  weight_wrong  EXCEPTION;
  score_wrong   EXCEPTION;
  n_score       scores.score%TYPE;
  n_weight      performance_parts.weight%TYPE;
  running_total NUMBER := 0;
  max_score     CONSTANT scores.score%TYPE := 9;
  max_weight    CONSTANT performance_parts.weight%TYPE:= 1;
BEGIN
  SELECT s.score INTO n_score
  FROM SCORES s
  WHERE evaluation_id = s.evaluation_id
  AND performance_id = s.performance_id;

  SELECT p.weight INTO n_weight
  FROM PERFORMANCE_PARTS p
  WHERE performance_id = p.performance_id;

  BEGIN
    IF (n_weight > max_weight) OR (n_weight < 0) THEN
      RAISE weight_wrong;
    END IF;
  END;

  BEGIN
    IF (n_score > max_score) OR (n_score < 0) THEN
      RAISE score_wrong;
    END IF;
  END;

  running_total := n_score * n_weight;
```

```
  RETURN running_total;

EXCEPTION
  WHEN weight_wrong THEN
    DBMS_OUTPUT.PUT_LINE(
      'The weight of a score must be between 0 and ' || max_weight);
    RETURN -1;
  WHEN score_wrong THEN
    DBMS_OUTPUT.PUT_LINE(
      'The score must be between 0 and ' || max_score);
    RETURN -1;
END calculate_score;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about user-defined exceptions
- *Oracle Database PL/SQL Language Reference* for more information about handling raised exceptions

# 8

# Using Triggers

This chapter contains the following topics:



## About Triggers

A **trigger** is a PL/SQL unit that is stored in the database and (if it is in the enabled state) automatically executes ("fires") in response to a specified event.

A trigger has this structure:

```
TRIGGER trigger_name
  triggering_event
  [ trigger_restriction ]
BEGIN
  triggered_action;
END;
```

The *trigger_name* must be unique for triggers in the schema. A trigger can have the same name as another kind of object in the schema (for example, a table); however, Oracle recommends using a naming convention that avoids confusion.

If the trigger is in the **enabled** state, the *triggering_event* causes the database to execute the *triggered_action* if the *trigger_restriction* is either TRUE or omitted. The *triggering_event* is associated with either a table, a view, a schema, or the database, and it is one of these:

- DML statement (described in "About Data Manipulation Language (DML) Statements" on page 5-1)
- DDL statement (described in "About Data Definition Language (DDL) Statements" on page 6-1)
- Database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN)

If the trigger is in the **disabled** state, the *triggering_event* does not cause the database to execute the *triggered_action*, even if the *trigger_restriction* is TRUE or omitted.

By default, a trigger is created in the enabled state. You can disable an enabled trigger, and enable a disabled trigger.

Unlike a subprogram, a trigger cannot be invoked directly. A trigger is invoked only by its triggering event, which can be caused by any user or application. You might be unaware that a trigger is executing unless it causes an error that is not handled properly.

A **simple trigger** can fire at exactly one of these **timing points**:

- Before the triggering event executes (statement-level `BEFORE` trigger)
- After the triggering event executes (statement-level `AFTER` trigger)
- Before each row that the event affects (row-level `BEFORE` trigger)
- After each row that the event affects (row-level `AFTER` trigger)

A **compound trigger** can fire at multiple timing points. For information about compound triggers, see *Oracle Database PL/SQL Language Reference*.

An **`INSTEAD OF` trigger** is defined on a view, and its triggering event is a DML statement. Instead of executing the DML statement, Oracle Database executes the `INSTEAD OF` trigger. For more information, see "Creating an INSTEAD OF Trigger" on page 8-5.

A **system trigger** is defined on a schema or the database. A trigger defined on a schema fires for each event associated with the owner of the schema (the current user). A trigger defined on a database fires for each event associated with all users.

One use of triggers is to enforce business rules that apply to all client applications. For example, suppose that data added to the `EMPLOYEES` table must have a certain format, and that many client applications can add data to this table. A trigger on the table can ensure the proper format of all data added to it. Because the trigger executes whenever any client adds data to the table, no client can circumvent the rules, and the code that enforces the rules can be stored and maintained only in the trigger, rather than in every client application. For other uses of triggers, see *Oracle Database PL/SQL Language Reference*.

> **See Also:** *Oracle Database PL/SQL Language Reference* for complete information about triggers

## Creating Triggers

To create triggers, use either the SQL Developer tool Create Trigger or the DDL statement `CREATE TRIGGER`. This topic shows how to use both of these ways to create triggers.

By default, a trigger is created in the enabled state. To create a trigger in disabled state, use the `CREATE TRIGGER` statement with the `DISABLE` clause.

> **Note:** To create triggers, you must have appropriate privileges; however, for this discussion and simple application, you do not need this additional information.

Topics:

- About OLD and NEW Pseudorecords
- Tutorial: Creating a Trigger that Logs Table Changes

- Tutorial: Creating a Trigger that Generates a Primary Key for a Row Before It Is Inserted
- Creating an INSTEAD OF Trigger
- Creating Triggers that Log LOGON and LOGOFF Events

**Note:** To do the tutorials in this document, you must be connected to Oracle Database as the user HR from SQL Developer. For instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6.

**See Also:**

- *Oracle Database SQL Developer User's Guide* for information about SQL Developer dialog boxes for creating objects
- *Oracle Database PL/SQL Language Reference* for more information about the CREATE TRIGGER statement
- "Editing Installation Script Files that Create Triggers" on page 10-6

## About OLD and NEW Pseudorecords

When a row-level trigger fires, the PL/SQL run-time system creates and populates the two pseudorecords OLD and NEW. They are called pseudorecords because they have some, but not all, of the properties of records.

For the row that the trigger is processing:

- For an INSERT trigger, OLD contains no values, and NEW contains the new values.
- For an UPDATE trigger, OLD contains the old values, and NEW contains the new values.
- For a DELETE trigger, OLD contains the old values, and NEW contains no values.

To reference a pseudorecord, put a colon before its name—:OLD or :NEW—as in Example 8–1.

**See Also:** *Oracle Database PL/SQL Language Reference* for more information about OLD and NEW pseudorecords

## Tutorial: Creating a Trigger that Logs Table Changes

This tutorial shows how to use the CREATE TRIGGER statement to create a trigger, EVAL_CHANGE_TRIGGER, which adds a row to the table EVALUATIONS_LOG whenever an INSERT, UPDATE, or DELETE statement changes the EVALUATIONS table.

The trigger adds the row *after* the triggering statement executes, and uses the **conditional predicates** INSERTING, UPDATING, and DELETING to determine which of the three possible DML statements fired the trigger.

EVAL_CHANGE_TRIGGER is a **statement-level trigger** and an **AFTER trigger**.

This trigger is part of the sample application that the tutorials and examples in this document show how to develop and deploy.

### To create EVALUATIONS_LOG and EVAL_CHANGE_TRIGGER:

1. Create the EVALUATIONS_LOG table:

```
CREATE TABLE EVALUATIONS_LOG ( log_date DATE
                             , action VARCHAR2(50));
```

2. Create EVAL_CHANGE_TRIGGER:

```
CREATE OR REPLACE TRIGGER EVAL_CHANGE_TRIGGER
  AFTER INSERT OR UPDATE OR DELETE
  ON EVALUATIONS
DECLARE
  log_action  EVALUATIONS_LOG.action%TYPE;
BEGIN
  IF INSERTING THEN
    log_action := 'Insert';
  ELSIF UPDATING THEN
    log_action := 'Update';
  ELSIF DELETING THEN
    log_action := 'Delete';
  ELSE
    DBMS_OUTPUT.PUT_LINE('This code is not reachable.');
  END IF;

  INSERT INTO EVALUATIONS_LOG (log_date, action)
    VALUES (SYSDATE, log_action);
END;
```

## Tutorial: Creating a Trigger that Generates a Primary Key for a Row Before It Is Inserted

The sequence EVALUATIONS_SEQ, created in "Creating and Managing Sequences" on page 6-20, generates primary keys for the EVALUATIONS table. However, these primary keys are not inserted into the table automatically.

This tutorial shows how to use the SQL Developer Create Trigger tool to create a trigger named NEW_EVALUATION_TRIGGER, which fires *before* a row is inserted into the EVALUATIONS table, and generates the unique number for the primary key of that row, using evaluations_seq. The trigger fires once *for each row* affected by the triggering INSERT statement.

NEW_EVALUATION_TRIGGER is a **row-level trigger** and a **BEFORE trigger**.

This trigger is part of the sample application that the tutorials and examples in this document show how to develop and deploy.

### To create the NEW_EVALUATION trigger:

1. On the Connections tab, expand **hr_conn**.

   Under the hr_conn icon, a list of schema object types appears.

2. Right-click **Triggers**.

   A list of choices appears.

3. Click **New Trigger**.

   The Create Trigger window opens. The field Schema has the value HR and the field Name has the default value TRIGGER1.

4. In the Name field, type NEW_EVALUATION_TRIGGER over the default value.

5. Click the tab **Trigger**.

   The Trigger pane appears. By default, the field Trigger Type has the value TABLE, the check box Enabled is selected, the field Table Owner has the value HR, the field Table Name has the value COUNTRIES, the options Before and Statement Level are

selected, the options After and Row Level are deselected, and the check boxes Insert, Update, and Delete are deselected.

6. In the field Table Name, from the drop-down menu, select **EVALUATIONS**.
7. Select the option **Row Level**.

   The option Statement Level is now deselected.
8. Select the check box **Insert**.
9. Click **OK**.

   The NEW_EVALUATION_TRIGGER pane opens, showing the CREATE TRIGGER statement that created the trigger:

```
CREATE OR REPLACE
TRIGGER NEW_EVALUATION_TRIGGER
BEFORE INSERT ON EVALUATIONS
FOR EACH ROW
BEGIN
  NULL;
END;
```

10. In the CREATE TRIGGER statement, replace NULL with this:

```
:NEW.evaluation_id := evaluations_seq.NEXTVAL
```

    The title of the NEW_EVALUATION_TRIGGER pane is in italic font, indicating that the trigger is not yet saved in the database.
11. From the File menu, select **Save**.

    Oracle Database compiles the procedure and saves it. The title of the NEW_EVALUATION_TRIGGER pane is no longer in italic font.

## Creating an INSTEAD OF Trigger

A view presents the output of a query as a table. If you want to change a view as you would change a table, you must create INSTEAD OF triggers. Instead of changing the view, they change the underlying tables.

For example, consider the view EMP_LOCATIONS, whose NAME column is created from the LAST_NAME and FIRST_NAME columns of the EMPLOYEES table:

```
CREATE VIEW EMP_LOCATIONS AS
SELECT e.EMPLOYEE_ID,
  e.LAST_NAME || ', ' || e.FIRST_NAME NAME,
  d.DEPARTMENT_NAME DEPARTMENT,
  l.CITY CITY,
  c.COUNTRY_NAME COUNTRY
FROM EMPLOYEES e, DEPARTMENTS d, LOCATIONS l, COUNTRIES c
WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID AND
 d.LOCATION_ID = l.LOCATION_ID AND
 l.COUNTRY_ID = c.COUNTRY_ID
ORDER BY LAST_NAME;
```

To update EMP_LOCATIONS.NAME, you must update EMPLOYEES.LAST_NAME and EMPLOYEES.FIRST_NAME. This is what the INSTEAD OF trigger in Example 8–1 does.

This trigger is part of the sample application that the tutorials and examples in this document show how to develop and deploy.

NEW and OLD are **pseudorecords** that the PL/SQL run-time engine creates and populates whenever a row-level trigger fires. OLD and NEW store the original and new values, respectively, of the record being processed by the trigger. They are called pseudorecords because they do not have all properties of PL/SQL records.

***Example 8–1 Creating an INSTEAD OF Trigger***

```
CREATE OR REPLACE TRIGGER update_name_view_trigger
INSTEAD OF UPDATE ON emp_locations
BEGIN
  UPDATE employees SET
    first_name = substr( :NEW.name, instr( :new.name, ',' )+2),
    last_name = substr( :NEW.name, 1, instr( :new.name, ',')-1)
  WHERE employee_id = :OLD.employee_id;
END;
```

**See Also:** *Oracle Database PL/SQL Language Reference*

## Creating Triggers that Log LOGON and LOGOFF Events

This tutorial shows how to use the CREATE TRIGGER statement to create two triggers, hr_logon_trigger and hr_logoff_trigger. *After* someone logs on as user HR, hr_logon_trigger adds a row to the table HR_USERS_LOG. *Before* someone logs off as user HR, hr_logoff_trigger adds a row to the table HR_USERS_LOG.

hr_logon_trigger and hr_logoff_trigger are **system triggers**. hr_logon_trigger is a **BEFORE trigger**, and hr_logoff_trigger is an **AFTER trigger**.

These triggers are *not* part of the sample application that the tutorials and examples in this document show how to develop and deploy.

### To create HR_USERS_LOG, HR_LOGON_TRIGGER, and HR_LOGOFF_TRIGGER:

1. Create the HR_USERS_LOG table:

```
CREATE TABLE hr_users_log (
  user_name VARCHAR2(30),
  activity VARCHAR2(20),
  event_date DATE
);
```

2. Create hr_logon_trigger:

```
CREATE OR REPLACE TRIGGER hr_logon_trigger
  AFTER LOGON
  ON HR.SCHEMA
BEGIN
  INSERT INTO hr_users_log (user_name, activity, event_date)
  VALUES (USER, 'LOGON', SYSDATE);
END;
```

3. Create hr_logoff_trigger:

```
CREATE OR REPLACE TRIGGER hr_logoff_trigger
  BEFORE LOGOFF
  ON HR.SCHEMA
BEGIN
  INSERT INTO hr_users_log (user_name, activity, event_date)
  VALUES (USER, 'LOGOFF', SYSDATE);
END;
```

## Changing Triggers

To change a trigger, use either the SQL Developer tool Edit or the DDL statement `CREATE TRIGGER` with the `OR REPLACE` clause.

**To change a standalone stored subprogram using the Edit tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Triggers**.

   A list of triggers appears.

3. Click the trigger to change.

   To the right of the Connections pane, a frame with appears. Its top tab has the name of the trigger to change. Under the top tab are subtabs.

4. Click the subtab **Code**.

   The Code pane appears. It shows the code that created the trigger to change.

5. On the Code pane, click the icon **Edit**.

   Another pane appears, also with the name of the trigger to change.

6. In the pane, change the code.

   The title of the pane is in italic font, indicating that the change is not yet saved in the database.

7. Select **Save** from the File menu.

   Oracle Database compiles the trigger and saves it. The title of the pane is no longer in italic font.

**See Also:**

- "About Data Definition Language (DDL) Statements" on page 6-1 for general information that applies to the `CREATE OR REPLACE TRIGGER` statement
- *Oracle Database PL/SQL Language Reference* for more information about the `CREATE OR REPLACE TRIGGER` statement

## Disabling and Enabling Triggers

You might need to temporarily disable triggers if they reference objects that are unavailable, or if you must upload a large amount of data without the delay that triggers cause (as in a recovery operation). After the referenced objects become available, or you have finished uploading the data, you can re-enable the triggers.

To disable or enable a single trigger, use the `ALTER TRIGGER` statement with the `DISABLE` or `ENABLE` clause. For example:

```
ALTER TRIGGER eval_change_trigger DISABLE;
ALTER TRIGGER eval_change_trigger ENABLE;
```

To disable or enable all triggers on a particular table, use the `ALTER TABLE` statement with the `DISABLE ALL TRIGGERS` or `ENABLE ALL TRIGGERS` clause. For example:

```
ALTER TABLE evaluations DISABLE ALL TRIGGERS;
ALTER TABLE evaluations ENABLE ALL TRIGGERS;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about the `ALTER TRIGGER` statement
- *Oracle Database SQL Language Reference* for more information about the `ALTER TABLE` statement

## About Trigger Compilation and Dependencies

Running a `CREATE TRIGGER` statement compiles the trigger being created. If this compilation causes an error, the `CREATE TRIGGER` statement fails. To see the compilation errors, run this statement:

```
SELECT * FROM USER_ERRORS WHERE TYPE = 'TRIGGER';
```

Compiled triggers depend on the schema objects on which they are defined. For example, `NEW_EVALUATION_TRIGGER` depends on the `EVALUATIONS` table:

```
CREATE OR REPLACE
TRIGGER NEW_EVALUATION_TRIGGER
BEFORE INSERT ON EVALUATIONS
FOR EACH ROW
BEGIN
  :NEW.evaluation_id := evaluations_seq.NEXTVAL;
END;
```

To see the schema objects on which triggers depend, run this statement:

```
SELECT * FROM ALL_DEPENDENCIES WHERE TYPE = 'TRIGGER';
```

If an object on which a trigger depends is dropped, or changed such that there is a mismatch between the trigger and the object, then the trigger is invalidated. The next time the trigger is invoked, it is recompiled. To recompile a trigger immediately, run the `ALTER TRIGGER` statement with the `COMPILE` clause. For example:

```
ALTER TRIGGER NEW_EVALUATION_TRIGGER COMPILE;
```

**See Also:**

- *Oracle Database PL/SQL Language Reference* for more information about trigger compilation and dependencies

## Dropping Triggers

You must drop a trigger before dropping the objects on which it depends.

To drop a trigger, use either the SQL Developer navigation frame and Drop tool, or the DDL statement `DROP TRIGGER`.

**To drop a trigger using the Drop tool:**

1. On the Connections tab, expand **hr_conn**.

   Under the `hr_conn` icon, a list of schema object types appears.

2. Expand **Triggers**.

   A list of triggers appears.

3. Right-click the name of the trigger to drop.

   A list of choices appears.

4. Click **Drop Trigger**.

   The Drop window opens.

5. Click **Apply**.

   The Confirmation window opens.

6. Click **OK**.

**See Also:**

- "About Data Definition Language (DDL) Statements" on page 6-1 for general information that applies to the `DROP TRIGGER` statement
- *Oracle Database PL/SQL Language Reference* for information about the `DROP TRIGGER` statement

# 9

# Working in a Global Environment

This chapter contains the following topics:



## About Globalization Support Features

Globalization support features enable you to develop multilingual applications that can be run simultaneously from anywhere in the world. An application can render the content of the user interface, and process data, using the native language and locale preferences of the user.

> **Note:** In the past, Oracle called globalization support **National Language Support (NLS)**, but NLS is actually a subset of globalization support. NLS is the ability to choose a national language and store data using a specific character set. NLS is implemented with NLS parameters.

Topics:



> **See Also:** *Oracle Database Globalization Support Guide* for more information about globalization support features

## About Language Support

Oracle Database enables you to store, process, and retrieve data in native languages. The languages that can be stored in a database are all languages written in scripts that are encoded by Oracle-supported character sets. Through the use of Unicode databases and datatypes, Oracle Database supports most contemporary languages.

Additional support is available for a subset of the languages. The database can, for example, display dates using translated month names, and can sort text data according to cultural conventions.

In this document, the term **language support** refers to the additional language-dependent functionality, and not to the ability to store text of a specific language. For example, language support includes displaying dates or sorting text according to specific locales and cultural conventions. Additionally, for some supported languages, Oracle Database provides translated error messages and a translated user interface for the database utilities.

**See Also:**

- "About the NLS_LANGUAGE Parameter" on page 9-10
- *Oracle Database Globalization Support Guide* for a complete list of languages that Oracle Database supports
- *Oracle Database Globalization Support Guide* for a list of languages into which Oracle Database messages are translated

## About Territory Support

Oracle Database supports cultural conventions that are specific to geographical locations. The default local time format, date format, and numeric and monetary conventions depend on the local territory setting. Setting different NLS parameters enables the database session to use different cultural settings. For example, you can set the euro (`EUR`) as the primary currency and the Japanese yen (`JPY`) as the secondary currency for a given database session, even when the territory is `AMERICA`.

**See Also:**

- "About the NLS_TERRITORY Parameter" on page 9-12
- *Oracle Database Globalization Support Guide* for a complete list of territories that Oracle Database supports

## About Date and Time Formats

Different countries have different conventions for displaying the hour, day, month, and year. For example:

| Country | Date Format | Example | Time Format | Example |
|---|---|---|---|---|
| China | `yyyy-mm-dd` | `2005-02-28` | `hh24:mi:ss` | `13:50:23` |
| Estonia | `dd.mm.yyyy` | `28.02.2005` | `hh24:mi:ss` | `13:50:23` |
| Germany | `dd.mm.rr` | `28.02.05` | `hh24:mi:ss` | `13:50:23` |
| UK | `dd/mm/yyyy` | `28/02/2005` | `hh24:mi:ss` | `13:50:23` |
| U.S. | `mm/dd/yyyy` | `02/28/2005` | `hh:mi:ssxff am` | `1:50:23.555 PM` |

**See Also:**

- "About the NLS_DATE_FORMAT Parameter" on page 9-14
- "About the NLS_DATE_LANGUAGE Parameter" on page 9-16
- "About NLS_TIMESTAMP_FORMAT and NLS_TIMESTAMP_TZ_FORMAT Parameters" on page 9-17
- *Oracle Database Globalization Support Guide* for information about date/time data types and time zone support
- *Oracle Database SQL Language Reference* for information about date and time formats

## About Calendar Formats

Oracle Database stores this calendar information for each territory:

- **First day of the week**

  Sunday in some cultures, Monday in others. Set by the `NLS_TERRITORY` parameter.

- **First week of the calendar year**

  Some countries use week numbers for scheduling, planning, and bookkeeping. In the ISO standard, this week number can differ from the week number of the calendar year. For example, 1st Jan 2005 is in ISO week number 53 of 2004. An ISO week starts on Monday and ends on Sunday. To support the ISO standard, Oracle Database provides the IW date format element, which returns the ISO week number. The first calendar week of the year is set by the `NLS_TERRITORY` parameter.

- **Number of days and months in a year**

  Oracle Database supports six calendar systems in addition to the Gregorian calendar, which is the default. These additional calendar systems are:

  - Japanese Imperial

    Has the same number of months and days as the Gregorian calendar, but the year starts with the beginning of each Imperial Era.

  - ROC Official

    Has the same number of months and days as the Gregorian calendar, but the year starts with the founding of the Republic of China.

  - Persian

    The first six months have 31 days each, the next five months have 30 days each, and the last month has either 29 days or (in leap year) 30 days.

  - Thai Buddha uses a Buddhist calendar.
  - Arabic Hijrah has 12 months and 354 or 355 days.
  - English Hijrah has 12 months and 354 or 355 days.

  The calendar system is specified by the `NLS_CALENDAR` parameter.

- **First year of era**

  The Islamic calendar starts from the year of the Hegira. The Japanese Imperial calendar starts from the beginning of an Emperor's reign (for example, 1998 is the tenth year of the Heisei era).

**See Also:**

- "About the NLS_TERRITORY Parameter" on page 9-12
- "About the NLS_CALENDAR Parameter" on page 9-18
- *Oracle Database Globalization Support Guide* for information about calendar formats

## About Numeric and Monetary Formats

Different countries have different numeric and monetary conventions. For example:

| Country | Numeric Format | Monetary Format |
|---|---|---|
| China | 1,234,567.89 | ¥1,234.56 |
| Estonia | 1 234 567,89 | 1 234,56 kr |
| Germany | 1.234.567,89 | 1.234,56€ |
| UK | 1,234,567.89 | £1,234.56 |
| U.S. | 1,234,567.89 | $1,234.56 |

**See Also:**

- "About the NLS_NUMERIC_CHARACTERS Parameter" on page 9-19
- "About the NLS_CURRENCY Parameter" on page 9-20
- "About the NLS_ISO_CURRENCY Parameter" on page 9-21
- "About the NLS_DUAL_CURRENCY Parameter" on page 9-22
- *Oracle Database Globalization Support Guide* for information about numeric and list parameters
- *Oracle Database Globalization Support Guide* for information about monetary parameters
- *Oracle Database SQL Language Reference* for information about number format models

## About Linguistic Sorting and String Searching

Different languages have different sort orders (collating sequences). Also, different countries or cultures that use the same alphabets sort words differently. For example, in Danish, Æ is after Z, and Y and Ü are considered to be variants of the same letter.

**See Also:**

- "About the NLS_SORT Parameter" on page 9-23
- "About the NLS_COMP Parameter" on page 9-24
- *Oracle Database Globalization Support Guide* for more information about linguistic sorting and string searching

## About Length Semantics

In single-byte character sets, the number of bytes and the number of characters in a string are the same. In multibyte character sets, a character or code point consists of

one or more bytes. Calculating the number of characters based on byte length can be difficult in a variable-width character set. Calculating column length in bytes is called **byte semantics**, while measuring column length in characters is called **character semantics**.

Character semantics is useful for specifying the storage requirements for multibyte strings of varying widths. For example, in a Unicode database (AL32UTF8), suppose that you must have a `VARCHAR2` column that can store up to five Chinese characters with five English characters. Using byte semantics, this column requires 15 bytes for the Chinese characters, which are 3 bytes long, and 5 bytes for the English characters, which are 1 byte long, for a total of 20 bytes. Using character semantics, the column requires 10 characters.

**See Also:**

- "About the NLS_LENGTH_SEMANTICS Parameter" on page 9-25
- *Oracle Database Globalization Support Guide* for information about character sets and length semantics

## About Unicode and SQL National Character Data Types

**Unicode** is a character encoding system that defines every character in most of the spoken languages in the world. In Unicode, every character has a unique code, regardless of the platform, program, or language.

You can store Unicode characters in an Oracle Database in two ways:

- You can create a Unicode database that enables you to store UTF-8 encoded characters as SQL character datatypes (`CHAR`, `VARCHAR2`, `CLOB`, and `LONG`).
- You can declare columns and variables that have SQL national character datatypes.

The **SQL national character data types** are `NCHAR`, `NVARCHAR2`, and `NCLOB`. They are also called **Unicode data types**, because they are used only for storing Unicode data.

The national character set, which is used for all SQL national character data types, is specified when the database is created. The national character set can be either UTF8 or AL16UTF16 (default).

When you declare a column or variable of the type `NCHAR` or `NVARCHAR2`, the length that you specify is the number of characters, not the number of bytes.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about Unicode
- *Oracle Database Globalization Support Guide* for more information about storing Unicode characters in an Oracle Database
- *Oracle Database Globalization Support Guide* for more information about SQL national character data types

# About Initial NLS Parameter Values

Except in SQL Developer, the initial values of NLS parameters are set by database initialization parameters. The DBA can set the values of initialization parameters in the initialization parameter file, and they take effect the next time the database is started.

In SQL Developer, the initial values of NLS parameters are as shown in Table 9–1.

*Table 9–1 Initial Values of NLS Parameters in SQL Developer*

| Parameter | Initial Value |
|---|---|
| NLS_CALENDAR | GREGORIAN |
| NLS_CHARACTERSET | AL32UTF8 |
| NLS_COMP | BINARY |
| NLS_CURRENCY | $ |
| NLS_DATE_FORMAT | DD-MON-RR |
| NLS_DATE_LANGUAGE | AMERICAN |
| NLS_DUAL_CURRENCY | $ |
| NLS_ISO_CURRENCY | AMERICA |
| NLS_LANGUAGE | AMERICAN |
| NLS_LENGTH_SEMANTICS | BYTE |
| NLS_NCHAR_CHARACTERSET | AL16UTF16 |
| NLS_NCHAR_CONV_EXCP | FALSE |
| NLS_NUMERIC_CHARACTERS | ., |
| NLS_SORT | BINARY |
| NLS_TERRITORY | AMERICA |
| NLS_TIMESTAMP_FORMAT | DD-MON-RR HH.MI.SSXFF AM |
| NLS_TIMESTAMP_TZ_FORMAT | DD-MON-RR HH.MI.SSXFF AM TZR |
| NLS_TIME_FORMAT | HH.MI.SSXFF AM |
| NLS_TIME_TZ_FORMAT | HH.MI.SSXFF AM TZR |

**See Also:** *Oracle Database Administrator's Guide* for information about initialization parameters and initialization parameter files

## Viewing NLS Parameter Values

To view the current values of NLS parameters, use either the SQL Developer report National Language Support Parameters or this query:

```
SELECT * FROM V$NLS_PARAMETERS;
```

### To view the National Language Support Parameters report:

1. In the SQL Developer window, click the tab **Reports**.

   The Reports pane shows a list of reports.

2. Expand **Data Dictionary Reports**.

   A list of data dictionary reports appears.

3. Expand **About Your Database**.

   A list of reports appears.

4. Click **National Language Support Parameters**.

   The Select Connection window opens. It has a Connection field with a drop-down menu.

5. From the drop-down menu, select **hr_conn**.
6. Click **OK**.

   The Select Connection window closes and the National Language Support Parameters pane appears, showing the names of the NLS parameters and their current values.

   **See Also:** *Oracle Database SQL Developer User's Guide* for more information about SQL Developer reports

## Changing NLS Parameter Values

You can change the value of one or more NLS parameters in any of these ways:

- Change the values for all SQL Developer connections, current and future.
- On the client, change the settings of the corresponding NLS environment variables.

  Only on the client, the new values of the NLS environment variables override the values of the corresponding NLS parameters.

  You can use environment variables to specify locale-dependent behavior for the client. For example, on a Linux system, this statement sets the value of the `NLS_SORT` environment variable to `FRENCH`, overriding the value of the `NLS_SORT` parameter:

  ```
  % setenv NLS_SORT FRENCH
  ```

  **Note:** Environment variables might be platform-dependent.

- Change the values only for the current session, using an `ALTER SESSION` statement with this syntax:

  ```
  ALTER SESSION SET parameter_name=parameter_value
    [ parameter_name=parameter_value ]... ;
  ```

  Only in the current session, the new values override those set in all of the preceding ways.

  You can use the `ALTER SESSION` to test your application with the settings for different locales.

- Change the values only for the current SQL function invocation.

  Only for the current SQL function invocation, the new values override those set in all of the preceding ways.

Topics:

- Changing NLS Parameter Values for All SQL Developer Connections
- Changing NLS Parameter Values for the Current SQL Function Invocation

**See Also:**

- *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` statement
- *Oracle Database Globalization Support Guide* for more information about setting NLS parameters

## Changing NLS Parameter Values for All SQL Developer Connections

The following procedure shows how to change the values of NLS parameters for all SQL Developer connections, current and future.

**To change National Language Support Parameter values:**

1. In the SQL Developer window, click the menu **Tools**.

   A drop-down menu appears.

2. From drop-down menu, select **Preferences**.

   The Preferences window opens.

3. In the Preferences window, expand **Database**.

   A list of database preferences appears.

4. Click **NLS Parameters**.

   A list of NLS Parameters and their current values appears. The value fields are drop-down menus.

5. From the drop-down menu to the right of each parameter whose value you want to change, select the desired value.

6. Click **OK**.

   The NLS parameters now have the values that you specified. To verify these values, see "Viewing NLS Parameter Values" on page 9-6.

   **See Also:** *Oracle Database SQL Developer User's Guide* for more information about SQL Developer preferences

## Changing NLS Parameter Values for the Current SQL Function Invocation

SQL functions whose behavior depends on the values of NLS parameters are called **locale-dependent**. Some locale-dependent SQL functions have optional NLS parameters. These functions are:

- `TO_CHAR`
- `TO_DATE`
- `TO_NUMBER`
- `NLS_UPPER`
- `NLS_LOWER`
- `NLS_INITCAP`
- `NLSSORT`

In all of the preceding functions, you can specify these NLS parameters:

- `NLS_DATE_LANGUAGE`
- `NLS_DATE_LANGUAGE`
- `NLS_NUMERIC_CHARACTERS`
- `NLS_CURRENCY`
- `NLS_ISO_CURRENCY`
- `NLS_DUAL_CURRENCY`

- NLS_CALENDAR
- NLS_SORT

In the NLSSORT function, you can also specify these NLS parameters:

- NLS_LANGUAGE
- NLS_TERRITORY
- NLS_DATE_FORMAT

To specify NLS parameters in a function, use this syntax:

```
'parameter=value' ['parameter=value']...
```

Suppose that you want NLS_DATE_LANGUAGE to be AMERICAN when this query is evaluated:

```
SELECT last_name FROM employees WHERE hire_date > '01-JAN-1999';
```

You can set NLS_DATE_LANGUAGE to AMERICAN before running the query:

```
ALTER SESSION SET NLS_DATE_LANGUAGE=American;
SELECT last_name FROM employees WHERE hire_date > '01-JAN-1999';
```

Alternatively, you can set NLS_DATE_LANGUAGE to AMERICAN inside the query, using the locale-dependent SQL function TO_DATE with its optional NLS_DATE_LANGUAGE parameter:

```
SELECT last_name FROM employees
WHERE hire_date > TO_DATE('01-JAN-1999', 'DD-MON-YYYY',
                          'NLS_DATE_LANGUAGE=AMERICAN');
```

**Tip:** Specify optional NLS parameters in locale-dependent SQL functions only in SQL statements that must be independent of the session NLS parameter values. Using session default values for NLS parameters in SQL functions usually results in better performance.

**See Also:** *Oracle Database Globalization Support Guide* for more information about locale-dependent SQL functions with optional NLS parameters

## About Individual NLS Parameters

Topics:

- About the NLS_ISO_CURRENCY Parameter
- About the NLS_DUAL_CURRENCY Parameter
- About the NLS_SORT Parameter
- About the NLS_COMP Parameter
- About the NLS_LENGTH_SEMANTICS Parameter

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about setting up a globalization support environment
- "Changing NLS Parameter Values" on page 9-7

## About Locale and the NLS_LANG Parameter

A **locale** is a linguistic and cultural environment in which a system or application runs. The simplest way to specify a locale for Oracle Database software is to set the `NLS_LANG` parameter.

The `NLS_LANG` parameter sets the default values of the parameters `NLS_LANGUAGE` and `NLS_TERRITORY` for both the server session (for example, SQL statement processing) and the client application (for example, display formatting in Oracle Database tools). The `NLS_LANG` parameter also sets the character set that the client application uses for data entered or displayed.

The default value of `NLS_LANG` is set during database installation. You can use the `ALTER SESSION` statement to change the values of NLS parameters, including those set by `NLS_LANG`, for your session. However, only the client can change the NLS settings in the client environment.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about specifying a locale with the `NLS_LANG` parameter
- *Oracle Database Globalization Support Guide* for information about languages, territories, character sets, and other locale data supported by Oracle Database
- "About the NLS_LANGUAGE Parameter" on page 9-10
- "About the NLS_TERRITORY Parameter" on page 9-12
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_LANGUAGE Parameter

**Specifies:** Default language of the database. Default conventions for:

- Language for server messages
- Language for names and abbreviations of days and months that are specified in the SQL functions `TO_CHAR` and `TO_DATE`
- Symbols for default-language equivalents of AM, PM, AD, and BC
- Default sorting order for character data when the `ORDER BY` clause is specified
- Writing direction
- Affirmative and negative response strings (for example, `YES` and `NO`)

**Acceptable Values:** Any language name that Oracle supports. For a list, see *Oracle Database Globalization Support Guide*.

**Default Value:** Set by NLS_LANG, described in "About Locale and the NLS_LANG Parameter" on page 9-10.

**Sets default values of:**

- NLS_DATE_LANGUAGE, described in "About the NLS_DATE_LANGUAGE Parameter" on page 9-16.
- NLS_SORT, described in "About the NLS_SORT Parameter" on page 9-23.

Example 9–1 shows how setting NLS_LANGUAGE to ITALIAN and GERMAN affects server messages and month abbreviations.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–1 NLS_LANGUAGE Affects Server Message and Month Abbreviations***

1. Note the current value of NLS_LANGUAGE.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value in step 1 is not ITALIAN, change it:

```
ALTER SESSION SET NLS_LANGUAGE=ITALIAN;
```

3. Query a nonexistent table:

```
SELECT * FROM nonexistent_table;
```

   Result:

```
SELECT * FROM nonexistent_table
              *
ERROR at line 1:
ORA-00942: tabella o vista inesistente
```

4. Run this query:

```
SELECT LAST_NAME, HIRE_DATE
FROM EMPLOYEES
WHERE EMPLOYEE_ID IN (111, 112, 113);
```

   Result:

```
LAST_NAME                 HIRE_DATE
------------------------- ---------
Sciarra                   30-SET-97
Urman                     07-MAR-98
Popp                      07-DIC-99

3 rows selected.
```

5. Change the value of NLS_LANGUAGE to GERMAN:

```
ALTER SESSION SET NLS_LANGUAGE=GERMAN;
```

6. Repeat the query from step 3.

   Result:

```
SELECT * FROM nonexistent_table
              *
ERROR at line 1:
ORA-00942: Tabelle oder View nicht vorhanden
```

7. Repeat the query from step 4.

   Result:

```
LAST_NAME                 HIRE_DATE
------------------------- ---------
Sciarra                   30-SEP-97
Urman                     07-MRZ-98
Popp                      07-DEZ-99

3 rows selected.
```

8. Set NLS_LANGUAGE to the value that it had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_LANGUAGE parameter
- "About Language Support" on page 9-2
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_TERRITORY Parameter

**Specifies:** Default conventions for:

- Date format
- Timestamp format
- Decimal character and group separator
- Local currency symbol
- ISO currency symbol
- Dual currency symbol

**Acceptable Values:** Any territory name that Oracle supports. For a list, see *Oracle Database Globalization Support Guide*.

**Default Value:** Set by NLS_LANG, described in "About Locale and the NLS_LANG Parameter" on page 9-10.

**Sets default values of:**

- NLS_DATE_FORMAT, described in "About the NLS_DATE_FORMAT Parameter" on page 9-14.
- NLS_TIMESTAMP_FORMAT and NLS_TIMESTAMP_TZ_FORMAT, described in "About NLS_TIMESTAMP_FORMAT and NLS_TIMESTAMP_TZ_FORMAT Parameters" on page 9-17.
- NLS_NUMERIC_CHARACTERS, described in "About the NLS_NUMERIC_CHARACTERS Parameter" on page 9-19.
- NLS_CURRENCY, described in "About the NLS_CURRENCY Parameter" on page 9-20.

- NLS_ISO_CURRENCY, described in "About the NLS_ISO_CURRENCY Parameter" on page 9-21.
- NLS_DUAL_CURRENCY, described in "About the NLS_DUAL_CURRENCY Parameter" on page 9-22.

Example 9–2 shows how setting NLS_TERRITORY to JAPAN and AMERICA affects the currency symbol.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–2 NLS_TERRITORY Affects Currency Symbol***

1. Note the current value of NLS_TERRITORY.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value in step 1 is not JAPAN, change it:

```
ALTER SESSION SET NLS_TERRITORY=JAPAN;
```

3. Run this query:

```
SELECT TO_CHAR(SALARY,'L99G999D99') SALARY
FROM EMPLOYEES
WHERE EMPLOYEE_ID IN (100, 101, 102);
```

   Result:

```
SALARY
--------------------
          ¥24,000.00
          ¥17,000.00
          ¥17,000.00

3 rows selected.
```

4. Change the value of NLS_TERRITORY to AMERICA:

```
ALTER SESSION SET NLS_TERRITORY=AMERICA;
```

5. Repeat the query from step 3.

   Result:

```
SALARY
--------------------
          $24,000.00
          $17,000.00
          $17,000.00

3 rows selected.
```

6. Set NLS_TERRITORY to the value that it had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_TERRITORY parameter
- "About Territory Support" on page 9-2
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_DATE_FORMAT Parameter

**Specifies:** Default date format to use with the TO_CHAR and TO_DATE functions (which are introduced in "Using Conversion Functions in Queries" on page 4-22).

**Acceptable Values:** Any any valid datetime format model. For example:

```
NLS_DATE_FORMAT='MM/DD/YYYY'
```

For information about datetime format models, see *Oracle Database SQL Language Reference*.

**Default Value:** Set by NLS_TERRITORY, described in "About the NLS_TERRITORY Parameter" on page 9-12.

The default date format might not correspond to the convention used in a given territory. To get dates in localized formats, you can use the 'DS' (short date) and 'DL' (long date) formats.

Example 9–3 shows how setting NLS_TERRITORY to AMERICA and FRANCE affects the default, short, and long date formats.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–3 NLS_TERRITORY Affects Date Formats***

1. Note the current value of NLS_TERRITORY.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value in step 1 is not AMERICA, change it:

   ```
   ALTER SESSION SET NLS_TERRITORY=AMERICA;
   ```

3. Run this query:

   ```
   SELECT hire_date "Default",
          TO_CHAR(hire_date,'DS') "Short",
          TO_CHAR(hire_date,'DL') "Long"
   FROM employees
   WHERE employee_id IN (111, 112, 113);
   ```

   Result:

   ```
   Default    Short      Long
   --------- ---------- ----------------------------
   30-SEP-97 9/30/1997  Tuesday, September 30, 1997
   07-MAR-98 3/7/1998   Saturday, March 07, 1998
   07-DEC-99 12/7/1999  Tuesday, December 07, 1999

   3 rows selected.
   ```

4. Change the value of NLS_TERRITORY to FRANCE:

```
ALTER SESSION SET NLS_TERRITORY=FRANCE;
```

5. Repeat the query from step 3.

   Result:

```
Default  Short      Long
-------- ---------- --------------------------
30/09/97 30/09/1997 tuesday 30 september 1997
07/03/98 07/03/1998 saturday 7 march 1998
07/12/99 07/12/1999 tuesday 7 december 1999

3 rows selected.
```

   (To get the names of the days and months in French, you must set either NLS_LANGUAGE or NLS_DATE_LANGUAGE to FRENCH before running the query.)

6. Set NLS_TERRITORY to the value that it had at step 1.

Example 9–4 changes the value of NLS_DATE_FORMAT, overriding the default value set by NLS_TERRITORY.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–4 NLS_DATE_FORMAT Overrides NLS_TERRITORY***

1. Note the current values of NLS_TERRITORY and NLS_DATE_FORMAT.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the valueof NLS_TERRITORY in step 1 is not AMERICA, change it:

```
ALTER SESSION SET NLS_TERRITORY=AMERICA;
```

3. If the valueof NLS_DATE_FORMAT in step 1 is not 'Day Month ddth', change it:

```
ALTER SESSION SET NLS_DATE_FORMAT='Day Month ddth';
```

4. Run this query (from previous example, step 3):

```
SELECT hire_date "Default",
       TO_CHAR(hire_date,'DS') "Short",
       TO_CHAR(hire_date,'DL') "Long"
FROM employees
WHERE employee_id IN (111, 112, 113);
```

   Result:

```
Default                 Short      Long
----------------------- ---------- ----------------------------
Tuesday   September 30th 9/30/1997  Tuesday, September 30, 1997
Saturday  March     07th 3/7/1998   Saturday, March 07, 1998
Tuesday   December  07th 12/7/1999  Tuesday, December 07, 1999

3 rows selected.
```

5. Set NLS_TERRITORY and NLS_DATE_FORMAT to the values that they had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_DATE_FORMAT parameter
- *Oracle Database SQL Language Reference* for more information about the TO_CHAR function
- *Oracle Database SQL Language Reference* for more information about the TO_DATE function
- "About Date and Time Formats" on page 9-2
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_DATE_LANGUAGE Parameter

**Specifies:** Language for names and abbreviations of days and months that are produced by:

- SQL functions TO_CHAR and TO_DATE (which are introduced in "Using Conversion Functions in Queries" on page 4-22)
- Default date format (set by NLS_DATE_FORMAT, described in "About the NLS_DATE_FORMAT Parameter" on page 9-14)
- Symbols for default-language equivalents of AM, PM, AD, and BC

**Acceptable Values:** Any language name that Oracle supports. For a list, see *Oracle Database Globalization Support Guide*.

**Default Value:** Set by NLS_LANGUAGE, described in "About the NLS_LANGUAGE Parameter" on page 9-10.

Example 9–5 shows how setting NLS_DATE_LANGUAGE to FRENCH and SWEDISH affects the displayed system date.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–5 NLS_DATE_LANGUAGE Affects Displayed SYSDATE***

1. Note the current value of NLS_DATE_LANGUAGE.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value of NLS_DATE_LANGUAGE in step 1 is not FRENCH, change it:

```
ALTER SESSION SET NLS_DATE_LANGUAGE=FRENCH;
```

3. Run this query:

```
SELECT TO_CHAR(SYSDATE, 'Day:Dd Month yyyy') "System Date"
FROM DUAL;
```

   Result:

```
System Date
--------------------------
Mercredi:01 Octobre   2008
```

4. Change the value of NLS_DATE_LANGUAGE to SWEDISH:

```
ALTER SESSION SET NLS_DATE_LANGUAGE=SWEDISH;
```

5. Repeat the query from step 3.

   Result:

```
System Date
-------------------------
Onsdag :01 Oktober   2008
```

6. Set NLS_DATE_LANGUAGE to the value that it had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_DATE_LANGUAGE parameter
- *Oracle Database SQL Language Reference* for more information about the TO_CHAR function
- *Oracle Database SQL Language Reference* for more information about the TO_DATE function
- "About Date and Time Formats" on page 9-2
- "Changing NLS Parameter Values" on page 9-7

## About NLS_TIMESTAMP_FORMAT and NLS_TIMESTAMP_TZ_FORMAT Parameters

**Specify:** Default date format for:

- TIMESTAMP datatype
- TIMESTAMP WITH LOCAL TIME ZONE datatype

**Acceptable Values:** Any any valid datetime format model. For example:

```
NLS_TIMESTAMP_FORMAT='YYYY-MM-DD HH:MI:SS.FF'
NLS_TIMESTAMP_TZ_FORMAT='YYYY-MM-DD HH:MI:SS.FF TZH:TZM'
```

For information about datetime format models, see *Oracle Database SQL Language Reference*.

**Default Value:** Set by NLS_TERRITORY, described in "About the NLS_TERRITORY Parameter" on page 9-12.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_TIMESTAMP_FORMAT parameter
- *Oracle Database Globalization Support Guide* for more information about the NLS_TIMESTAMP_TZ_FORMAT parameter
- *Oracle Database Globalization Support Guide* for information about date/time data types and time zone support
- *Oracle Database SQL Language Reference* for more information about the TIMESTAMP datatype
- *Oracle Database SQL Language Reference* for more information about the TIMESTAMP WITH LOCAL TIME ZONE data type
- "About Date and Time Formats" on page 9-2
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_CALENDAR Parameter

**Specifies:** Calendar system for the database.

**Acceptable Values:** Any calendar system that Oracle supports. For a list, see *Oracle Database Globalization Support Guide*.

**Default Value:** `Gregorian`

Example 9–6 shows how setting `NLS_CALENDAR` to `'English Hijrah'` and `Gregorian` affects the displayed system date.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–6 NLS_CALENDAR Affects Displayed SYSDATE***

1. Note the current value of `NLS_CALENDAR`.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value of `NLS_CALENDAR` in step 1 is not `'English Hijrah'`, change it:

   ```
   ALTER SESSION SET NLS_CALENDAR='English Hijrah';
   ```

3. Run this query:

   ```
   SELECT SYSDATE FROM DUAL;
   ```

   Result:

   ```
   SYSDATE
   -------------------------
   29 Ramadan           1429
   ```

4. Change the value of `NLS_CALENDAR` to `'Gregorian'`:

   ```
   ALTER SESSION SET NLS_CALENDAR='Gregorian';
   ```

5. Run this query:

   ```
   SELECT SYSDATE FROM DUAL;
   ```

   Result:

   ```
   SYSDATE
   ---------
   30-SEP-08
   ```

6. Set `NLS_CALENDAR` to the value that it had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the `NLS_CALENDAR` parameter
- "About Calendar Formats" on page 9-3
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_NUMERIC_CHARACTERS Parameter

**Specifies:** Decimal character (which separates the integer and decimal parts of a number) and group separator (which separates integer groups to show thousands and millions, for example). The group separator is the character returned by the numeric format element `G`.

**Acceptable Values:** Any two different single-byte characters except:

- A numeric character
- Plus (+)
- Minus (-)
- Less than (<)
- Greater than (>)

**Default Value:** Set by `NLS_TERRITORY`, described in "About the NLS_TERRITORY Parameter" on page 9-12.

In a SQL statement, you can represent a number as either:

- Numeric literal

  A numeric literal is not enclosed in quotation marks, always uses a period (.) as the decimal character, and never contains a group separator.

- Text literal

  A text literal is enclosed in single quotation marks. It is implicitly or explicitly converted to a number, if required, according to the current NLS settings.

Example 9–7 shows how two different `NLS_NUMERIC_CHARACTERS` settings affect the displayed result of the same query.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–7 NLS_NUMERIC_CHARACTERS Affects Decimal Character and Group Separator***

1. Note the current value of `NLS_NUMERIC_CHARACTERS`.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value of `NLS_NUMERIC_CHARACTERS` in step 1 is not `", ."` (decimal character is comma and group separator is period), change it:

   ```
   ALTER SESSION SET NLS_NUMERIC_CHARACTERS=",.";
   ```

3. Run this query:

   ```
   SELECT TO_CHAR(4000, '9G999D99') "Number" FROM DUAL;
   ```

   Result:

   ```
   Number
   ---------
    4.000,00
   ```

4. Change the value of `NLS_NUMERIC_CHARACTERS` to `". ,"` (decimal character is period and group separator is comma):

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS=".,";
```

5. Run this query:

```
SELECT TO_CHAR(4000, '9G999D99') "Number" FROM DUAL;
```

Result:

```
Number
---------
 4,000.00
```

6. Set NLS_NUMERIC_CHARACTERS to the value that it had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_NUMERIC_CHARACTERS parameter
- "About Numeric and Monetary Formats" on page 9-4
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_CURRENCY Parameter

**Specifies:** Local currency symbol (the character string returned by the numeric format element L).

**Acceptable Values:** Any valid currency symbol string.

**Default Value:** Set by NLS_TERRITORY, described in "About the NLS_TERRITORY Parameter" on page 9-12.

Example 9–8 changes the value of NLS_CURRENCY, overriding the default value set by NLS_TERRITORY.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–8 NLS_CURRENCY Overrides NLS_TERRITORY***

1. Note the current values of NLS_TERRITORY and NLS_CURRENCY.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the valueof NLS_TERRITORY in step 1 is not AMERICA, change it:

```
ALTER SESSION SET NLS_TERRITORY=AMERICA;
```

3. Run this query:

```
SELECT TO_CHAR(salary, 'L099G999D99') "Salary"
FROM EMPLOYEES
WHERE salary > 13000;
```

Result:

```
Salary
---------------------
          $024,000.00
          $017,000.00
          $017,000.00
          $014,000.00
```

```
          $013,500.00
```

4. Change the value of NLS_CURRENCY to '¥':

```
ALTER SESSION SET NLS_CURRENCY='¥';
```

5. Run this query:

```
SELECT TO_CHAR(salary, 'L099G999D99') "Salary"
FROM EMPLOYEES
WHERE salary > 13000;
```

Result:

```
Salary
--------------------
          ¥024,000.00
          ¥017,000.00
          ¥017,000.00
          ¥014,000.00
          ¥013,500.00
```

6. Set NLS_TERRITORY and NLS_CURRENCY to the values that they had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_CURRENCY parameter
- "About Numeric and Monetary Formats" on page 9-4
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_ISO_CURRENCY Parameter

**Specifies:** ISO currency symbol (the character string returned by the numeric format element C).

**Acceptable Values:** Any valid currency symbol string.

**Default Value:** Set by NLS_TERRITORY, described in "About the NLS_TERRITORY Parameter" on page 9-12.

Local currency symbols can be ambiguous, but ISO currency symbols are unique.

Example 9–9 shows that the territories AUSTRALIA and AMERICA have the same local currency symbol, but different ISO currency symbols.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–9 NLS_ISO_CURRENCY***

1. Note the current values of NLS_TERRITORY and NLS_ISO_CURRENCY.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value of NLS_TERRITORY in step 1 is not AUSTRALIA, change it:

```
ALTER SESSION SET NLS_TERRITORY=AUSTRALIA;
```

3. Run this query:

```
SELECT TO_CHAR(salary, 'L099G999D99') "Local",
       TO_CHAR(salary, 'C099G999D99') "ISO"
FROM EMPLOYEES
WHERE salary > 15000;
```

Result:

```
Local                ISO
-------------------- ------------------
          $024,000.00      AUD024,000.00
          $017,000.00      AUD017,000.00
          $017,000.00      AUD017,000.00
```

4. Change the value of `NLS_TERRITORY` to `AMERICA`:

```
ALTER SESSION SET NLS_TERRITORY=AMERICA;
```

5. Run this query:

```
SELECT TO_CHAR(salary, 'L099G999D99') "Local",
       TO_CHAR(salary, 'C099G999D99') "ISO"
FROM EMPLOYEES
WHERE salary > 15000;
```

Result:

```
Local                ISO
-------------------- ------------------
          $024,000.00      USD024,000.00
          $017,000.00      USD017,000.00
          $017,000.00      USD017,000.00
```

6. Set `NLS_TERRITORY` and `NLS_ISO_CURRENCY` to the values that they had at step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the `NLS_ISO_CURRENCY` parameter
- "About Numeric and Monetary Formats" on page 9-4
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_DUAL_CURRENCY Parameter

**Specifies:** Dual currency symbol (introduced to support the euro currency symbol during the euro transition period).

**Acceptable Values:** Any valid currency symbol string.

**Default Value:** Set by `NLS_TERRITORY`, described in "About the NLS_TERRITORY Parameter" on page 9-12.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the `NLS_DUAL_CURRENCY` parameter
- "About Numeric and Monetary Formats" on page 9-4
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_SORT Parameter

**Specifies:** Linguistic sort order (collating sequence) for queries that have the `ORDER BY` clause.

**Acceptable Values:**

- `BINARY`

  Sort order is based on the binary sequence order of either the database character set or the national character set, depending on the data type.

- Any linguistic sort name that Oracle supports

  Sort order is based on the order of the specified linguistic sort name. The linguistic sort name is usually the same as the language name, but not always. For a list of supported linguistic sort names, see *Oracle Database Globalization Support Guide*.

**Default Value:** Set by `NLS_LANGUAGE`, described in "About the NLS_LANGUAGE Parameter" on page 9-10.

Example 9–10 shows how two different `NLS_SORT` settings affect the displayed result of the same query. The settings are `BINARY` and Traditional Spanish (`SPANISH_M`). Traditional Spanish treats ch, ll, and ñ as letters that follow c, l, and n, respectively.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–10 NLS_SORT Affects Linguistic Sort Order***

1. Create table for Spanish words:

```
CREATE TABLE temp (name VARCHAR2(15));
```

2. Populate table with some Spanish words:

```
INSERT INTO temp (name) VALUES ('laguna');
INSERT INTO temp (name) VALUES ('llama');
INSERT INTO temp (name) VALUES ('loco');
```

3. Note the current value of `NLS_SORT`.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

4. If the valueof `NLS_SORT` in step 3 is not `BINARY`, change it:

```
ALTER SESSION SET NLS_SORT=BINARY;
```

5. Run this query:

```
SELECT * FROM temp ORDER BY name;
```

   Result:

```
NAME
---------------
laguna
llama
loco
```

6. Change the value of `NLS_SORT` to `SPANISH_M` (Traditional Spanish):

```
ALTER SESSION SET NLS_SORT=SPANISH_M;
```

7. Repeat the query from step 5.

   Result:

```
NAME
---------------
laguna
loco
llama
```

8. Drop the table:

```
DROP TABLE temp;
```

9. Set NLS_SORT to the value that it had at step 3.

### Case-Insensitive and Accent-Insensitive Sorts

Operations inside Oracle Database are sensitive to the case and the accents of the characters. To perform a case-insensitive sort, append _CI to the value of the NLS_SORT parameter (for example, BINARY_CI or XGERMAN_CI). To perform a sort that is both case-insensitive and accent-insensitive, append _AI to the value of the NLS_SORT parameter (for example, BINARY_AI or FRENCH_M_AI).

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_SORT parameter
- *Oracle Database Globalization Support Guide* for more information about case-insensitive and accent-insensitive sorts
- "About Linguistic Sorting and String Searching" on page 9-4
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_COMP Parameter

**Specifies:** Character comparison behavior of SQL operations.

**Acceptable Values:**

- BINARY

  SQL compares the binary codes of characters. One character is greater than another if it has a higher binary code.

- LINGUISTIC

  SQL performs a linguistic comparison based on the value of the NLS_SORT parameter, described in "About the NLS_SORT Parameter" on page 9-23.

- ANSI

  This value is provided only for backward compatibility.

**Default Value:** BINARY

Example 9–11 shows that the result of a query can depend on the NLS_COMP setting.

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–11 NLS_COMP Affects SQL Character Comparison***

1. Note the current values of NLS_SORT and NLS_COMP.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the values of NLS_SORT and NLS_COMP in step 1 are not SPANISH_M (Traditional Spanish) and BINARY, respectively, change them:

```
ALTER SESSION SET NLS_SORT=SPANISH_M NLS_COMP=BINARY;
```

3. *Run this query:

```
SELECT LAST_NAME FROM EMPLOYEES
WHERE LAST_NAME LIKE 'C%';
```

   Result:

```
LAST_NAME
-------------------------
Cabrio
Cambrault
Cambrault
Chen
Chung
Colmenares

6 rows selected
```

4. Change the value of NLS_COMP to LINGUISTIC:

```
ALTER SESSION SET NLS_COMP=LINGUISTIC;
```

5. Repeat the query from step 3.

   Result:

```
LAST_NAME
-------------------------
Cabrio
Cambrault
Cambrault
Colmenares

4 rows selected
```

   This time, Chen and Chung are not returned because Traditional Spanish treats ch as a single character that follows c.

6. Set NLS_SORT and NLS_COMP to the values that they had in step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_COMP parameter
- "About Linguistic Sorting and String Searching" on page 9-4
- "Changing NLS Parameter Values" on page 9-7

## About the NLS_LENGTH_SEMANTICS Parameter

**Specifies:** Length semantics for columns of the character data types CHAR, VARCHAR2, and LONG; that is, whether these columns are specified in bytes or in characters. (Applies only to columns that are declared after the parameter is set.)

**Acceptable Values:**

- BYTE

  New CHAR, VARCHAR2, and LONG columns are specified in bytes.

- CHAR

  New CHAR, VARCHAR2, and LONG columns are specified in characters.

**Default Value:** BYTE

To try this example in SQL Developer, enter the statements and queries in the SQL Worksheet. For information about the SQL Worksheet, see "Running Queries in SQL Developer" on page 4-2. The results shown here are from SQL*Plus; their format is slightly different in SQL Developer.

***Example 9–12 NLS_LENGTH_SEMANTICS Affects Storage of VARCHAR2 Column***

1. Note the current values of NLS_LENGTH_SEMANTICS.

   For instructions, see "Viewing NLS Parameter Values" on page 9-6.

2. If the value of NLS_LENGTH_SEMANTICS in step 1 is not BYTE, change it:

   ```
   ALTER SESSION SET NLS_LENGTH_SEMANTICS=BYTE;
   ```

3. Create a table with a VARCHAR2 column:

   ```
   CREATE TABLE SEMANTICS_BYTE(SOME_DATA VARCHAR2(20));
   ```

4. Click the tab **Connections**.

   The Connections pane shows the connection hr_conn.

5. Expand **hr_conn**.

   A list of schema object types appears, including Tables.

6. Expand **Tables**.

   A list of tables appears, including SEMANTICS_BYTE.

7. Select **SEMANTICS_BYTE**.

   To the right of the Connections pane, the Columns pane shows that for Column Name SOME_DATA, the Data Type is VARCHAR2(20 BYTE).

8. Change the value of NLS_LENGTH_SEMANTICS to CHAR:

   ```
   ALTER SESSION SET NLS_LENGTH_SEMANTICS=CHAR;
   ```

9. Create another table with a VARCHAR2 column:

   ```
   CREATE TABLE SEMANTICS_CHAR(SOME_DATA VARCHAR2(20));
   ```

10. In the Connections pane, click the **Refresh icon**.

    The list of tables now includes SEMANTICS_CHAR.

11. Select **SEMANTICS_CHAR**.

    The Columns pane shows that for Column Name SOME_DATA, the Data Type is VARCHAR2(20 CHAR).

12. Select **SEMANTICS_BYTE** again.

The Columns pane shows that for Column Name SOME_DATA, the Data Type is still VARCHAR2(20 BYTE).

13. Set the value of NLS_LENGTH_SEMANTICS to the value that it had in step 1.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the NLS_LENGTH_SEMANTICS parameter
- "About Length Semantics" on page 9-4
- "Changing NLS Parameter Values" on page 9-7

# Using Unicode in Globalized Applications

You can insert and retrieve Unicode data. Data is transparently converted among the database and client programs, which ensures that client programs are independent of the database character set and national character set.

Topics:

- Representing Unicode String Literals in SQL and PL/SQL
- Avoiding Data Loss During Character-Set Conversion

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about SQL and PL/SQL programming with Unicode
- *Oracle Database Globalization Support Guide* for general information about programming with Unicode

## Representing Unicode String Literals in SQL and PL/SQL

There are three ways to represent a Unicode string literal in SQL or PL/SQL:

- N'*string*'

  Example: N'résumé'.

  Limitations: See "Avoiding Data Loss During Character-Set Conversion" on page 9-28.

- NCHR(*number*)

  The SQL function NCHR returns the character whose binary equivalent is *number* in the national character set. The character returned has data type NVARCHAR2.

  Example: NCHR(36) represents $ in the default national character set, AL16UTF16.

  Limitations: Portability of the value of NCHR(*number*) is limited to applications that use the same national character set.

- UNISTR('*string*')

  The SQL function UNISTR converts *string* to the national character set.

  For portability and data preservation, Oracle recommends that *string* contain only ASCII characters and Unicode encoding values. A Unicode encoding value has the form \xxxx, where xxxx is the hexadecimal value of a character code value in UCS-2 encoding format.

  Example: UNISTR('G\0061ry') represents 'Gary'

ASCII characters are converted to the database character set and then to the national character set. Unicode encoding values are converted directly to the national character set.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about Unicode string literals
- *Oracle Database SQL Language Reference* for more information about the `NCHR` function
- *Oracle Database SQL Language Reference* for more information about the `UNISTR` function

## Avoiding Data Loss During Character-Set Conversion

As part of a SQL or PL/SQL statement, a literal (with or without the prefix `N`) is encoded in the same character set as the rest of the statement. On the client side, the statement is encoded in the client character set, which is determined by the `NLS_LANG` parameter. On the server side, the statement is encoded in the database character set.

When the SQL or PL/SQL statement is transferred from the client to the database, its character set is converted accordingly. If the database character set does not contain all characters that the client used in the text literals, then data is lost in this conversion. `NCHAR` string literals are more vulnerable than `CHAR` text literals, because they are designed to be independent of the database character set.

To avoid data loss in conversion to an incompatible database character set, you can activate the `NCHAR` literal replacement functionality. For more information, see *Oracle Database Globalization Support Guide*.

# 10

# Deploying an Oracle Database Application

This chapter contains the following topics:



## About Deployment Environments

**Deployment** is installing your application in one or more environments where other users can run it.

The schema in which you develop your application is called the **development environment**. (The development environment for the sample application is the sample schema `HR`.) The other environments in which you deploy your application are called **deployment environments**. These environments might exist in your organization; if not, you can create them.

The first deployment environment is the **test environment**. In the test environment, you can thoroughly test the functionality of the application, determine whether it is packaged correctly, and fix any problems before deploying it in the **production environment**.

You might also deploy your application to an **education environment**, either before or after deploying it to the production environment. An education environment provides a place for users to practice running the application without affecting other environments.

## About Installation Script Files

To deploy an application, you run one or more installation script files. If these files do not exist, you can create them, with SQL Developer or any text editor.

An **installation script file** is an executable file (`.sql` file) that contains an installation script. An **installation script** is composed ot DDL statements, `INSERT` statements, or both. When you run your installation script files, the DDL statements create the schema objects of your application in the deployment environment, and the `INSERT` statements insert the data from the tables in your development environment (the

**source tables**) into the corresponding tables in the deployment environment (the **new tables**).

Topics:



## About DDL Statements and Schema Object Dependencies

When you run your installation script files, their DDL statements create the schema objects of your application in the deployment environment. To create the installation script files correctly and run them in the correct order, you must understand the dependencies between the schema objects of your application.

If the definition of object A references object B, then A depends on B. Therefore, you must create B before you create A. Otherwise, the statement that creates B either fails or creates B in an invalid state, depending on the object type.

Typically, you install schema objects and data in the deployment environment in this order:

1. Package specifications
2. Tables (with constraints and indexes) in correct order
3. Sequences (often used by triggers)
4. Triggers
5. Synonyms
6. Views (which might reference functions, procedures, or synonyms)
7. Package bodies
8. Data

However, for a complex application, the order for creating the objects is rarely obvious. Usually, you must consult the database designer or a diagram of the design.

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for more information about schema object dependencies
- "About Data Definition Language (DDL) Statements" on page 6-1

## About INSERT Statements and Constraints

When you run your installation script files, their `INSERT` statements insert the data from the source tables into the corresponding new tables. For each source table in your application, you must determine whether any constraints could be violated when their data is inserted in the new table. If so, you must first disable those constraints, then insert the data, and then try to re-enable the constraints. If a data item violates a constraint, you cannot re-enable that constraint until you correct the data item.

If you are simply inserting lookup data in correct order, constraints are not violated. Therefore, you do not need to disable them first.

If you are inserting data from an outside source (such as a file, spreadsheet, or older application), or from many tables that have much dependent data, disable the constraints before inserting the data.

Some possible ways to disable and re-enable the constraints are:

- Using SQL Developer, disable and re-enable the constraints one at a time.
- Edit the installation script file, adding SQL statements that disable and re-enable each constraint.
- Create a SQL script with SQL statements that disable and enable each constraint.
- Find the constraints in the Oracle Database data dictionary, and create a SQL script with the SQL statements to disable and enable each constraint.

  To find and enable the constraints used in the `EVALUATIONS`, `PERFORMANCE_PARTS`, and `SCORES` tables, enter these statements into a SQL Worksheet window:

```
SELECT 'ALTER TABLE '|| TABLE_NAME || ' DISABLE CONSTRAINT '||
  CONSTRAINT_NAME ||';'
  FROM user_constraints
 WHERE table_name IN ('EVALUATIONS','PERFORMANCE_PARTS','SCORES');

SELECT 'ALTER TABLE '|| TABLE_NAME || ' ENABLE CONSTRAINT '||
  CONSTRAINT_NAME ||';'
  FROM user_constraints
 WHERE table_name IN ('EVALUATIONS','PERFORMANCE_PARTS','SCORES');
```

**See Also:**

- "About the INSERT Statement" on page 5-2
- "Ensuring Data Integrity in Tables" on page 6-5

## Creating Installation Script Files

This topic explains how to use SQL Developer to create an installation script file, when and how to edit installation script files that create sequences and triggers, and how to create installation script files for the schema objects and data of the sample application.

The tutorials in this topic assume that you created the objects in the sample schema `HR`, using the instructions in this document, and are deploying the sample application in another standard `HR` schema.

---

**Note:** To do the tutorials in this document, you must be connected to Oracle Database as the user `HR` from SQL Developer. For instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6.

---

Topics:

## Creating an Installation Script File with SQL Developer

To create an installation script file with SQL Developer, use the Database Export tool. You specify the name of the installation script file and the objects and data to export, and SQL Developer generates DDL statements for creating the objects and `INSERT` statements for inserting the data into new tables, and writes these statements to the installation script file.

> **Note:** In the following procedure, you might have to enlarge the SQL Developer windows to see all fields and options.

**To create an installation script file with the Database Export tool:**

1. If you have not done so, create a directory for the installation script file, separate from the Oracle Database installation directory (for example, `C:\my_exports`).
2. In the SQL Developer window, click the menu **Tools**.

   A drop-down menu appears.
3. From drop-down menu, select **Database Export**.

   The Source/Destination window opens.
4. In the Source/Destination window:
   1. In the File field, type the full path name of the installation script file (for example, `C:\my_exports\hr_export.sql`).

      The file name must end with `.sql`.
   2. From the Connections drop-down menu, select your connection (for example, `hr_conn`).
   3. Select the desired DDL Options (and deselect any selected undesired options).

      By default, Terminator and Pretty Print are selected and all other options are deselected. If you deselect Terminator, the installation script file fails.

      For descriptions of the DDL Options, see *Oracle Database SQL Developer User's Guide*.
   4. Click **Next**.

   The Types to Export window appears, listing the types of objects, and data, that you can export. To the left of each object is a check box. By default, every check box is selected.
5. In the Types to Export window:
   1. Deselect the check boxes for the types that you do *not* want to export.

      Selecting or deselecting Toggle All selects or deselects all check boxes.
   2. If you do not want to export data, deselect the check box **Data**.
   3. Click **Next**.

   The Specify Objects window appears.
6. In the Specify Objects window:
   1. In the drop-down menu with the value `HR`, accept that value.
   2. Click **Go**.

A list appears, showing all objects in schema HR whose types you specified in step 5, and, if you specified Data in step 5, names of tables in schema HR.

3. To export all items on the list, click >>; otherwise, select the objects to export and the tables with data to export, and click >.

   The items to be exported move from the original column to the other column. (To move all of them back to the original column, click <<; to move individual items back, select them and click <.)

4. Click **Next**.

The Export Summary window appears.

7. In the Export Summary window, click **Finish**.

   The Exporting window opens, showing that exporting is occurring. When exporting is complete, the Exporting window closes, and the SQL Worksheet shows the contents of the installation script file that you specified in step 4.

8. In the installation script file, check that:

   - Referenced objects are created before their dependent objects.
   - Tables are created before data is inserted into them.

   If necessary, edit the file in the SQL Worksheet or any text editor.

**See Also:**

- "About Data Definition Language (DDL) Statements" on page 6-1
- "About the INSERT Statement" on page 5-2
- *Oracle Database SQL Developer User's Guide* for more information about SQL Developer dialog boxes
- *Oracle Database Advanced Application Developer's Guide* for information about schema object dependencies
- "Tutorial: Creating an Installation Script File for the Sequence and Tables" on page 10-6
- "Tutorial: Creating an Installation Script File for the Package" on page 10-8
- "Tutorial: Creating an Installation Script File for the Synonym and View" on page 10-9
- "Tutorial: Creating an Installation Script File for the Data" on page 10-11

## Editing Installation Script Files that Create Sequences

For a sequence, SQL Developer generates a CREATE SEQUENCE statement whose START WITH value is relative to the current value of the sequence in the development environment.

If your application uses the sequence to generate unique keys, and you *will not* insert the data from the source tables into the corresponding new tables, then you might want to edit the START WITH value in the installation script file. You can edit the installation script file in the SQL Worksheet or any text editor.

**See Also:** "Tutorial: Creating a Sequence" on page 6-21

## Editing Installation Script Files that Create Triggers

If your application has a BEFORE INSERT trigger on a source table, and you *will* insert the data from that source table into the corresponding new table, then you must decide if you want the trigger to fire before each INSERT statement in the installation script file inserts data into the new table.

For example, in the sample application, NEW_EVALUATION_TRIGGER fires before a row is inserted into the EVALUATIONS table, and generates the unique number for the primary key of that row, using EVALUATIONS_SEQ. The trigger fires once for each row affected by the triggering INSERT statement. (NEW_EVALUATION_TRIGGER is created in "Tutorial: Creating a Trigger that Generates a Primary Key for a Row Before It Is Inserted" on page 8-4.)

The source EVALUATIONS table is populated with primary keys. If you do not want the installation script to put new primary key values in the new EVALUATIONS table, then you must edit the CREATE TRIGGER statement in the installation script file as shown in bold:

```
CREATE OR REPLACE
TRIGGER NEW_EVALUATION_TRIGGER
BEFORE INSERT ON EVALUATIONS
FOR EACH ROW
BEGIN
  IF :NEW.evaluation_id IS NULL THEN
    :NEW.evaluation_id := evaluations_seq.NEXTVAL
  END IF;
END;
```

Also, check the current value of the sequence. If it not is greater than the maximum value in the primary key column, make it greater.

You can edit the installation script file in the SQL Worksheet or any text editor.

Two alternatives to editing the installation script file are:

- Change the trigger definition in the source file and then re-create the installation script file.

  For information about changing triggers, see "Changing Triggers" on page 8-7.

- Disable the trigger before running the data installation script file, and then re-enable it afterward.

  For information about disabling and enabling triggers, see "Disabling and Enabling Triggers" on page 8-7.

If your application uses triggers to populate auditing columns (such as CREATED_ON and CREATED_BY), you must preserve your source table data by having the trigger set a new value only if the current value is null, with code like this:

```
IF :NEW.CREATED_ON IS NULL THEN
  :NEW.CREATED_ON := SYSDATE;
END IF;
```

**See Also:** "Creating Triggers" on page 8-2

## Tutorial: Creating an Installation Script File for the Sequence and Tables

This tutorial shows how to use the SQL Developer tool Database Export to create an installation script file for the tables and sequence of the sample application, including the constraints, indexes, and triggers associated with the tables.

**Note:** In the following procedure, you might have to enlarge the SQL Developer windows to see all fields and options.

**To create an installation script file for the tables and sequence:**

1. If you have not done so, create the directory `C:\my_exports`.
2. In the SQL Developer window, click the menu **Tools**.

   A drop-down menu appears.
3. From drop-down menu, select **Database Export**.

   The Source/Destination window opens.
4. In the Source/Destination window:
   1. In the File field, type: `C:\my_exports\2day_tables.sql`.
   2. From the Connections drop-down menu, select **hr_conn**.
   3. Accept the default DDL Options, Terminator and Pretty Print.
   4. Click **Next**.

   The Types to Export window appears.
5. In the Types to Export window:
   1. Deselect all check boxes *except* Tables, Sequences, Indexes, Constraints, and Triggers.
   2. Click **Next**.

   The Specify Objects window appears.
6. In the Specify Objects window:
   1. In the drop-down menu with the value `HR`, accept that value.
   2. In the drop-down menu with the value `All`, select the value **TABLE**.
   3. Click **Go**.

      A list of the tables in schema `HR` appears.
   4. Select **HR**.**PERFORMANCE_PARTS**, **HR**.**EVALUATIONS**, **HR**.**SCORES**, and **HR**.**EVALUATIONS_LOG**.
   5. Click **>**.

      The selected tables move from the original column to the other column.
   6. In the drop-down menu with the value `TABLE`, select the value **SEQUENCE**.
   7. Click **Go**.

      A list of the sequences in schema `HR` appears.
   8. Select **HR**.**EVALUATIONS_SEQ**.
   9. Click **>**.

      `HR.EVALUATIONS_SEQ` moves from the original column to the other column.
   10. (Optional) In the drop-down menu with the value `SEQUENCE`, select the value **All**.

HR.PERFORMANCE_PARTS, HR.EVALUATIONS, HR.SCORES, HR.EVALUATIONS_LOG and HR.EVALUATIONS_SEQ show.

11. Click **Next**.

    The Export Summary window appears.

7. In the Export Summary window, click **Finish**.

   The Exporting window opens, showing that exporting is occurring. When exporting is complete, the Exporting window closes, and the SQL Worksheet shows the contents of the file `C:\my_exports\2day_tables.sql`.

8. In the installation script file, `C:\my_exports\2day_tables.sql`, check that referenced objects are created before their dependent objects:

   - The sequence EVALUATIONS_SEQ must be created before the table EVALUATIONS, because EVALUATIONS has a trigger, NEW_EVALUATION_TRIGGER, that uses EVALUATIONS_SEQ.
   - The table EVALUATIONS_LOG must be created before the table EVALUATIONS, because EVALUATIONS has a trigger, EVAL_CHANGE_TRIGGER, that uses EVALUATIONS_LOG.
   - The tables EVALUATIONS and PERFORMANCE_PARTS must be created before the table SCORES, because SCORES has foreign keys to both EVALUATIONS and PERFORMANCE_PARTS.

   If necessary, edit the file in the SQL Worksheet or any text editor.

**See Also:**

- "Creating Tables" on page 6-3
- "Tutorial: Creating a Sequence" on page 6-21
- "Creating an Installation Script File with SQL Developer" on page 10-4

## Tutorial: Creating an Installation Script File for the Package

This tutorial shows how to use the SQL Developer tool Database Export to create an installation script file for the package (specification and body) of the sample application.

**Note:** In the following procedure, you might have to enlarge the SQL Developer windows to see all fields and options.

### To create an installation script file for the package:

1. If you have not done so, create the directory `C:\my_exports`.
2. In the SQL Developer window, click the menu **Tools**.

   A drop-down menu appears.

3. From drop-down menu, select **Database Export**.

   The Source/Destination window opens.

4. In the Source/Destination window:

   1. In the File field, type: `C:\my_exports\2day_package.sql`.

2. From the Connections drop-down menu, select **hr_conn**.
3. Accept the default DDL Options, Terminator and Pretty Print.
4. Click **Next**.

The Types to Export window appears.

5. In the Types to Export window:
   1. Deselect all check boxes *except* Package Spec and Package Body.
   2. Click **Next**.

   The Specify Objects window appears.
6. In the Specify Objects window:
   1. In the drop-down menu with the value `HR`, accept that value.
   2. In the drop-down menu with the value `All`, accept that value.
   3. Click **Go**.

      A list of the packages and package bodies in schema `HR` appears.
   4. If the only items listed are the package `HR.EMP_EVAL` and the package body `HR.EMP_EVAL`, click **>>**; otherwise, select those two items from the list and click **>**.

      The package `HR.EMP_EVAL` and the package body `HR.EMP_EVAL` move from the original column to the other column.
   5. Click **Next**.

   The Export Summary window appears.
7. In the Export Summary window, click **Finish**.

   The Exporting window opens, showing that exporting is occurring. When exporting is complete, the Exporting window closes, and the SQL Worksheet shows the contents of the installation script file that you specified in step 4.

**See Also:**

- "Tutorial: Creating a Package Specification" on page 7-12
- "Tutorial: Creating a Package Body" on page 7-14
- "Creating an Installation Script File with SQL Developer" on page 10-4

## Tutorial: Creating an Installation Script File for the Synonym and View

This tutorial shows how to use the SQL Developer tool Database Export to create an installation script file for the synonym and view of the sample application.

**Note:** In the following procedure, you might have to enlarge the SQL Developer windows to see all fields and options.

**To create an installation script file for the synonym and view:**

1. If you have not done so, create the directory `C:\my_exports`.
2. In the SQL Developer window, click the menu **Tools**.

   A drop-down menu appears.

3. From drop-down menu, select **Database Export**.

   The Source/Destination window opens.

4. In the Source/Destination window:
   1. In the File field, type: `C:\my_exports\2day_other.sql`.
   2. From the Connections drop-down menu, select **hr_conn**.
   3. Accept the default DDL Options, Terminator and Pretty Print.
   4. Click **Next**.

   The Types to Export window appears.

5. In the Types to Export window:
   1. Deselect all check boxes *except* Views and Synonyms.
   2. Click **Next**.

   The Specify Objects window appears.

6. In the Specify Objects window:
   1. In the drop-down menu with the value `HR`, accept that value.
   2. In the drop-down menu with the value `All`, select the value **VIEW**.
   3. Click **Go**.

      A list of the views in schema `HR` appears.
   4. Select **HR**.**EMP_LOCATIONS**.
   5. Click **>**.

      `HR.EMP_LOCATIONS` moves from the original column to the other column.
   6. In the drop-down menu with the value `VIEW`, select the value **SYNONYM**.
   7. Click **Go**.

      A list of the synonyms in schema `HR` appears.
   8. Select **HR**.**EMP**.
   9. Click **>**.

      `HR.EMP` moves from the original column to the other column.
   10. (Optional) In the drop-down menu with the value `SYNONYM`, select the value **All**.

       `HR.EMP_LOCATIONS` and `HR.EMP` show.
   11. Click **Next**.

   The Export Summary window appears.

7. In the Export Summary window, click **Finish**.

   The Exporting window opens, showing that exporting is occurring. When exporting is complete, the Exporting window closes, and the SQL Worksheet shows the contents of the file `C:\my_exports\2day_other.sql`.

**See Also:**

- "Creating Views" on page 6-17
- "Creating Synonyms" on page 6-23
- "Creating an Installation Script File with SQL Developer" on page 10-4

## Tutorial: Creating an Installation Script File for the Data

This tutorial shows how to use the SQL Developer tool Database Export to create an installation script file for the data of the sample application. If you followed the instructions in this document, you added data only to the table `PERFORMANCE_PARTS`.

---

**Note:** In the following procedure, you might have to enlarge the SQL Developer windows to see all fields and options.

---

**To create an installation script file for the data:**

1. If you have not done so, create the directory `C:\my_exports`.
2. In the SQL Developer window, click the menu **Tools**.

   A drop-down menu appears.
3. From drop-down menu, select **Database Export**.

   The Source/Destination window opens.
4. In the Source/Destination window:
   1. In the File field, type: `C:\my_exports\2day_data.sql`.
   2. From the Connections drop-down menu, select **hr_conn**.
   3. Accept the default DDL Options, Terminator and Pretty Print.
   4. Click **Next**.

   The Types to Export window appears.
5. In the Types to Export window:
   1. Deselect all check boxes *except* Data.
   2. Click **Next**.

   The Specify Data window appears.
6. In the Specify Data window:
   1. In the drop-down menu with the value `HR`, accept that value.
   2. Click **Go**.

      A list of the tables in schema `HR` appears.
   3. Select **HR.PERFORMANCE_PARTS**.
   4. Click **>**.

      `HR.PERFORMANCE_PARTS` moves from the original column to the other column.
   5. Click **Next**.

The Export Summary window appears.

7. In the Export Summary window, click **Finish**.

   The Exporting window opens, showing that exporting is occurring. When exporting is complete, the Exporting window closes, and the SQL Worksheet shows the contents of the file `C:\my_exports\2day_data.sql`.

**See Also:**

- "About the INSERT Statement" on page 5-2
- "Creating an Installation Script File with SQL Developer" on page 10-4

## Installing the Sample Application

To install the sample application, you run the installation script files that you created in the tutorials in "Creating Installation Script Files" on page 10-3, in this order:

1. `2day_tables.sql`
2. `2day_package.sql`
3. `2day_other.sql`
4. `2day_data.sql`

You can either run the files one at a time, or you can create and run a master SQL script file that runs them in order.

The following master SQL script runs the files in order, commits the changes, and writes server messages to a log file:

```
spool C:\my_exports\create_log.txt
@C:\my_exports\2day_tables.sql
@C:\my_exports\2day_package.sql
@C:\my_exports\2day_other.sql
@C:\my_exports\2day_data.sql
commit;
spool off
```

The SQL*Plus command `@`*`file_name`*`.sql` runs a file.

Typically, you run the master script file in SQL*Plus. However, if the master script specifies the full path names of the individual files (as in the preceding example), you can run it in SQL Developer.

The following procedure uses SQL Developer to install the sample application by running the installation script files one at a time, in order.

---

**Note:** The deployment environment must be different from the development environment, and is assumed to be another standard `HR` schema.

---

### To install the sample application using SQL Developer:

1. Connect to Oracle Database as user `HR` in the deployment environment.

   For instructions, see "Connecting to Oracle Database as User HR from SQL Developer" on page 2-6. For Connection Name, enter a name other than `hr_conn` (for example, `hr_conn_2`).

2. In the navigation frame, click the tab **Files**.

   The Files pane appears, showing the directories on your computer.

3. Navigate to the directory `C:\my_exports`.

4. Expand **my_exports**.

   The list of installation script files appears.

5. Right-click **2day_tables.sql**.

   A list of choices appears.

6. Select **Open**.

   In the SQL Worksheet, a new `2day_tables.sql` pane appears, showing the content of the `2day_tables.sql` file.

7. Click the icon **Run Script**.

   The Select Connection window opens.

8. For Connection, on the drop-down menu, select the connection to the deployment environment (created in step 1).

9. Click **OK**.

   The result of each statement in `2day_tables.sql` is displayed in the Script Output pane.

10. Click the icon **Clear**.

    The title of the `2day_tables.sql` pane is now in italic font, indicating that the changes have not been committed, and the SQL Worksheet is blank.

11. Repeat steps 5 through 10 for `2day_package.sql`.

12. Repeat steps 5 through 10 for `2day_other.sql`.

13. Repeat steps 5 through 10 for `2day_data.sql`.

14. Click the icon **Commit**.

    The changes are committed.

When you are sure that the individual installation script files run without errors, you can create a master SQL script (`.sql`) file that runs them in order, commits the changes, and writes the results to a log file. For example:

```
spool C:\my_exports\create_log.txt
@C:\my_exports\2day_tables.sql
@C:\my_exports\2day_package.sql
@C:\my_exports\2day_other.sql
@C:\my_exports\2day_data.sql
commit;
spool off
```

Typically, you run the master file in SQL*Plus. However, if the master file specifies the full path names of the individual files (as in the preceding example), you can open and run it in SQL Developer.

**See Also:**

- *Oracle Database SQL Developer User's Guide* for more information about running scripts in SQL Developer
- *SQL*Plus User's Guide and Reference* for more information about using scripts in SQL*Plus

## Checking the Validity of an Installation

After installing your application in a deployment environment, you can check its validity in the following ways in SQL Developer:

- In the Connections pane:
  1. Expand the connection to the deployment environment.
  2. Examine the definitions of the new objects.
- In the Reports pane:
  1. Expand **Data Dictionary Reports**.

     A list of data dictionary reports appears.
  2. Expand **All Objects**.

     A list of objects reports appears.
  3. Select **All Objects**.
  4. Click the icon **Run Report**.

     The Enter Bind Values window appears.
  5. Select either **Owner** or **Object**.
  6. Click **Apply**.

     The message "Executing Report" shows, followed by the report itself.

     For each object, this report lists the Owner, Object Type, Object Name, Status (Valid or Invalid), Date Created, and Last DDL. Last DDL is the date of the last DDL operation that affected the object.
  7. Select **Invalid Objects**.

     The Enter Bind Values window appears.
  8. Click **Apply**.

     The message "Executing Report" shows, followed by the report itself.

     For each object whose Status is Invalid, this report lists the Owner, Object Type, and Object Name.

     **See Also:** *Oracle Database SQL Developer User's Guide* for more information about SQL Developer reports

## Archiving the Installation Script Files

After verifying that the installation of your application is valid, you might want to archive your installation script files in a source code control system. Before doing so, add comments to each file, documenting its creation date and purpose. If you ever must deploy the same application to another environment, you can use these archived files.

**See Also:** *Oracle Database Utilities* for information about Oracle Data Pump, which enables very high-speed movement of data and metadata from one database to another

# Index

## Symbols



## A



## B

## C

## D



## E

## F



## G



## H



## I

## J



## K



## L



## M



## N

## O



## P



## Q



## R

## S

## T

## U



## V



## W