Oracle® Database Express Edition

2 Day Developer Guide

10*g* Release 2 (10.2)

B25108-01

February 2006

Oracle Database Express Edition 2 Day Developer Guide, 10*g* Release 2 (10.2)

B25108-01

Contributors: Eric Belden, Bjorn Engsig, Nancy Greenberg, Christopher Jones, Simon Law, Mark Townsend

Contents

List of Examples

Preface

This guide explains basic concepts behind development with Oracle Database Express Edition (Oracle Database XE) and provides examples on how to use basic language features of SQL and PL/SQL. This guide is intended to be a very basic introduction to development and references are provided in the text to detailed information about subjects.

This section contains the following topics:

Documentation Topics

This guide contains the following topics:

Title	Description
Overview of Development	Provides an overview of application development with Oracle Database Express Edition.
Managing Database Objects	Discusses creating and managing database objects in your schema, plus design considerations when developing applications with the Oracle Database XE. Also discusses the datatypes used with database objects in Oracle Database XE.
Using SQL	Describes how to use SQL with Oracle Database XE, including how to retrieve and manipulate data, use SQL functions, and create database objects.
Using PL/SQL	Describes the PL/SQL language, which can be used to develop applications for use with Oracle Database XE.
Using Procedures, Functions, and Packages	Describes how to develop procedures, functions, and packages with PL/SQL for use with Oracle Database XE.
Using Triggers	Discusses the development of triggers with PL/SQL code and the use of database triggers with Oracle Database XE.
Working in a Global Environment	Discusses how to develop applications in a globalization support environment, providing information for SQL and PL/SQL Unicode programming in a global environment.

Title	Description
Using SQL Command Line	Provides an introduction to SQL Command Line (SQL*Plus), an interactive and batch command-line query tool that is installed with Oracle Database XE.
Reserved Words	Lists the Oracle Database XE SQL and PL/SQL reserved words and keywords.
Using a PL/SQL Procedure With PHP	Provides an example of the use of a PL/SQL procedure with PHP.
Using a PL/SQL Procedure With JDBC	Rrovides an example of the use of a PL/SQL procedure with Java and JDBC.

Audience

This guide is intended for anyone interested in learning about the Oracle Database Express Edition development environment.It is primarily an introduction to application development for beginning developers.

To use this guide, you need to have a general understanding of relational database concepts as well as an understanding of the operating system environment under which you are running the Oracle Database XE.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. 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.

Related Documentation

For more information, see these Oracle resources:

- *Oracle Database Express Edition Getting Started Guide*
- *Oracle Database Express Edition Installation Guide for Linux*

- *Oracle Database Express Edition Installation Guide for Microsoft Windows*
- *Oracle Database Express Edition 2 Day DBA*
- *Oracle Database Express Edition Application Express User's Guide*
- *Oracle Database Express Edition 2 Day Plus Application Express Developer Guide*
- *Oracle Database Express Edition 2 Day Plus PHP Developer Guide*
- *Oracle Database Express Edition 2 Day Plus Java Developer Guide*
- *Oracle Database Express Edition 2 Day Plus .NET Developer Guide*
- *Oracle Database Express Edition ISV Embedding Guide*
- *Oracle Database Application Developer's Guide - Fundamentals*
- *Oracle Database Concepts*
- *Oracle Database SQL Reference*
- *Oracle Database PL/SQL User's Guide and Reference*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database Globalization Support Guide*

For the most recent version of the Oracle Database Express Edition documentation, see the Oracle Database XE online library:

`http://www.oracle.com/technology/xe/documentation`

Printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://www.oracle.com/technology/membership/`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://www.oracle.com/technology/documentation/`

For information about additional books

`http://www.oracle.com/technology/books/10g_books.html`

Conventions

The following text conventions are used in this document:

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

1 Overview of Development

This section provides an overview of developing applications with Oracle Database Express Edition (Oracle Database XE).

This section contains the following topics:

See Also: *Oracle Database Application Developer's Guide - Fundamentals* for a complete overview of application development

Overview of Developing Applications With Oracle Database XE

This section introduces you to developing applications with Oracle Database Express Edition.

This section contains the following topics:

See Also:

- *Oracle Database Express Edition Getting Started Guide* for a short tutorial to get you quickly up and running using Oracle Database XE
- *Oracle Database Express Edition 2 Day DBA* for information about getting started with Oracle Database XE
- Oracle Database XE home page on Oracle Technology Network:

  `http://www.oracle.com/technology/xe`
- Oracle Database XE Documentation Library:

  `http://www.oracle.com/technology/xe/documentation`
- Oracle Database XE Discussion forum:

  `http://www.oracle.com/technology/products/database/xe/forum.html`

Oracle Database Express Edition

Oracle Database Express Edition is a relational database that stores and retrieves collections of related information. A database, also called a database server, is the key to solving the problems of information management. In a relational database, collections of related information are organized into structures called tables. Each table contains rows (records) that are composed of columns (fields). The tables are stored in the database in structures called schemas, which are logical structures of data where database users store their tables.

The `HR` sample schema that is included with Oracle Database XE is an example of a schema with related tables. In the `HR` sample schema, there are tables to store information about employees and departments. The tables contain common columns that allow data from one table to be related to another. For each employee in the `employees` table, the department name in the `departments` table can be retrieved based on the department ID column that is in both tables. See "Sample HR Account" on page 1-5.

See Also:

- *Oracle Database Sample Schemas* for a description of the `HR` sample schema
- *Oracle Database Concepts* for an introduction to Oracle databases

SQL

Structured Query Language (SQL) is a nonprocedural programming language that enables you to access a relational database. Using SQL statements, you can query tables to display data, create objects, modify objects, and perform administrative tasks. All you need to do is describe in SQL what you want done, and the SQL language compiler automatically generates a procedure to navigate the database and perform the desired task.

For information about using SQL, see Chapter 3, "Using SQL".

See Also: *Oracle Database SQL Reference* for information about SQL

PL/SQL

PL/SQL is an Oracle procedural extension to SQL, the standard database access language. It is an advanced programming language, which like SQL, has a built-in treatment of the relational database domain. Applications written in any of the Oracle programmatic interfaces can call stored procedures and functions written in PL/SQL.

In PL/SQL, you can manipulate data with SQL statements and control program flow with procedural constructs such as loops. You can also do the following:

- Declare constants and variables
- Define procedures and functions
- Use collections and object types
- Trap runtime errors
- Create functions, packages, procedures, and triggers

For information about using PL/SQL, see Chapter 4, "Using PL/SQL".

See Also:

- *Oracle Database PL/SQL User's Guide and Reference* for information about PL/SQL
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL packages that are supplied with Oracle Database XE
- `http://www.oracle.com/technology/tech/pl_sql/index.html` for additional PL/SQL information and code samples for on the Oracle Technology Network (OTN)

Database Objects

You need to create database objects before you start developing your application. These database objects are primarily used to store and organize the data that your application manipulates. These databases objects include tables, indexes, views, sequences and synonyms.

When creating some database objects, you need to specify a datatype for the data that is used by the object. When you create a table, you must specify a datatype for each of its columns. A datatype associates a fixed set of properties with the values that can be used in a column, or in an argument of a procedure or function. Each column value and constant in a SQL statement has a datatype, which is associated with a specific storage format, constraints, and a valid range of values. The most common datatypes are character, numeric, and date.

For information about managing database objects, see Chapter 2, "Managing Database Objects".

Basic Application Development Concepts

This section discusses the basic concepts in application development with Oracle Database Express Edition.

- User interface

  The interface that your application displays to end users depends on the technology behind the application as well as the needs of the users themselves.

The Oracle Database XE browser-based user interface is an example of an application interface. See "Development Tools" on page 1-4.

- Client/server model

  In a traditional client/server program, your application code runs on a different machine than where the database (server) is located. Database calls are transmitted from this client machine to a database, such as Oracle Database XE. Data is transmitted from the client to the server for insert and update operations and returned from the server to the client for query operations where the data is processed on the client machine.

- Server-side coding

  You can develop application logic that resides entirely inside the database by using PL/SQL triggers that execute automatically when changes occur in the database or stored PL/SQL procedures or functions that are called explicitly. Off-loading the work from your application lets you reuse code that performs verification and cleanup and control database operations from a variety of clients.

Development Tools

There are various tools that you can use to develop with SQL and PL/SQL, and to manage database objects.

Oracle Database XE has a browser-based user interface for administering database objects, running SQL statements, PL/SQL code, building Web-based applications, and more. The starting point for this interface is the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.

You can also use SQL Command Line (SQL*Plus) to enter SQL statements and PL/SQL code. To use SQL Command Line, see Appendix A, "Using SQL Command Line".

Logging in to the Database Home Page

To log in to the Database Home Page:

1. Access the Database Home Page.

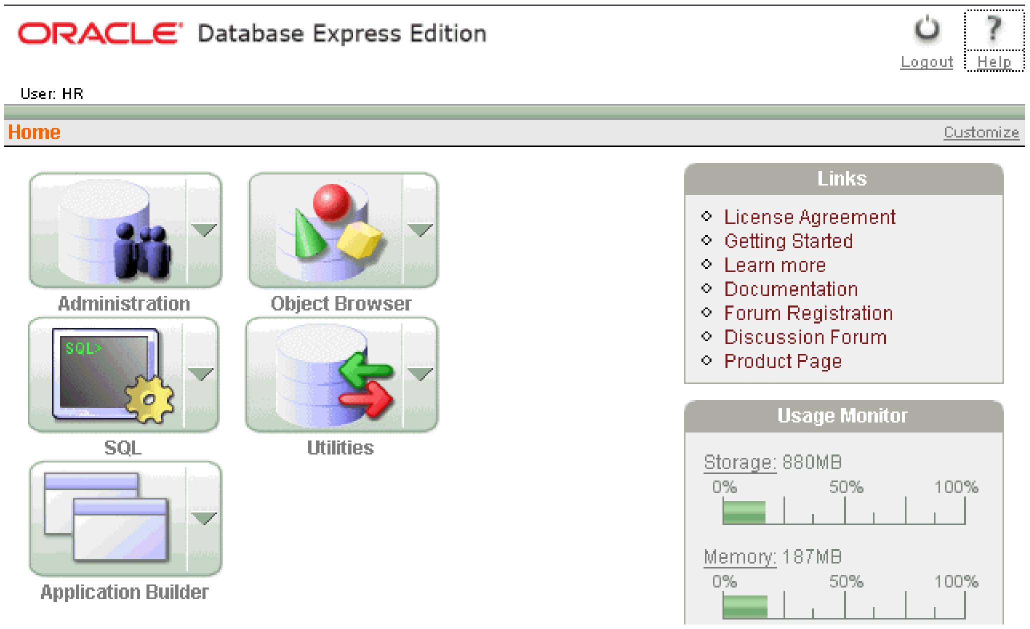
   The page can be accessed from your graphical desktop or pointing your Web browser to a specific URL. See "Accessing the Database Home Page" in *Oracle Database Express Edition 2 Day DBA*.

   To view the database objects or run the examples discussed in this guide, log in to the Database Home Page as the user `HR`. In the **Username** field enter `HR` and in the **Password** field enter your password for the `HR` user account, then click the **Login** button. See "Sample HR Account" on page 1-5.

   Note: If the `HR` user account is locked, you need to log in as a user with administrator privileges and unlock the account. When unlocking the account, ensure the HR user has both `CONNECT` and `RESOURCE` roles enabled. See "Locking and Unlocking User Accounts" in *Oracle Database Express Edition 2 Day DBA*.

2. On the Database Home Page, click the icon for the specific tool that you want to use. There are icons for **Administration**, **Application Builder**, **Object Browser**, **SQL**, and **Utilities** when you log in as the `HR` user.

The Database Home Page includes links to the **License Agreement**, **Getting Started**, **Learn More**, **Documentation**, **Forum Registration**, **Discussion Forum**, and **Product Page**. The Usage Monitor on the page provides information about the storage and memory use, number of sessions and users, and log archiving status.



See Also:

- *Oracle Database Express Edition Getting Started Guide* for a short tutorial to get you quickly up and running using Oracle Database XE
- *Oracle Database Express Edition 2 Day DBA* for information about getting started with Oracle Database Express Edition
- *Oracle Database Express Edition Application Express User's Guide* for a description of the icons on Database Home Page.

Sample HR Account

Oracle Database XE provides the `HR` sample user account for use with the examples in this guide. This `HR` user account is also referred to as the `HR` schema. A schema is a logical container for the database objects that the user creates.

The HR sample account is set up to be a simple Human Resources division for tracking information about the employees and the facilities. In the `HR` schema, each employee has an identification number, e-mail address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary. Each job has an identification code that associates it with a job title, a minimum salary, and a maximum salary for the job.

Each employee is assigned to a department, and each department is identified either by a unique department number or a short name. Each department is associated with one location, and each location has a full address that includes the street name, postal code, city, state or province, and the country code.

For information about viewing the database objects in the `HR` schema, including the structure of the `HR` tables, see "Managing Database Objects With Object Browser" on page 2-2.

See Also: *Oracle Database Sample Schemas* for information about the `HR` sample schema

Other Development Environments

This section lists other development languages that can be used with Oracle Database Express Edition. These environments are discussed in other guides.

This section contains the following topics:

Oracle Call Interface and Oracle C++ Call Interface

The Oracle Call Interface (OCI) and Oracle C++ Call Interface (OCCI) are native C and C++ APIs for accessing Oracle Database Express Edition from C and C++ applications.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information about OCI
- *Oracle C++ Call Interface Programmer's Guide* for more information about OCCI

Open Database Connectivity

Open Database Connectivity (ODBC) is a database access API that enables you to connect to a database and then prepare and run SQL statements against Oracle Database XE. In conjunction with an ODBC driver, an application can access any data source including data stored in spreadsheets, such as an Excel spreadsheet.

An Oracle ODBC driver is provided for both Windows and Linux (32 bit). The Oracle ODBC Driver conforms to ODBC 3.51 specifications. It supports all core APIs and a set of Level 1 and Level 2 functions. For Windows, the Driver Manager component is supplied by Microsoft. For Unix platforms, the Oracle Database XE driver has been tested using the latest Driver Manager available from `http://www.unixODBC.org`.

See Also:

- *Oracle Services for Microsoft Transaction Server Developer's Guide* for information about how to use the Oracle ODBC driver on Windows
- *Oracle Database Administrator's Reference 10g Release 2 (10.2) for UNIX-Based Operating Systems* for information about how to use the Oracle ODBC driver on Linux

Oracle Provider for OLE DB

OLE DB is an open standard data access methodology that uses a set of Component Object Model (COM) interfaces to access and manipulate different types of data. These interfaces are available from various database providers.

Oracle Provider for OLE DB (OraOLEDB) is an OLE DB data provider that offers high performance and efficient access to Oracle data by OLE DB consumers.

With the advent of the .NET framework, support is provided for using the OLEDB.NET Data Provider with OraOLEDB. With the correct connection attribute setting, an OLEDB.NET Data Provider can utilize OraOLEDB to access Oracle Database.

See Also: *Oracle Provider for OLE DB Developer's Guide* for information about developing applications to access Oracle Database XE using Oracle Provider for OLE DB

Oracle Data Provider for .NET

Oracle Data Provider for .NET (ODP.NET) provides data access for client applications from within Oracle Database XE. Oracle Data Provider for .NET is an implementation of a .NET data provider for Oracle Database, using and inheriting from classes and interfaces available in the Microsoft .NET Framework Class Library.

Following the .NET Framework, ODP.NET uses the ADO.NET model, which enables native providers to expose provider-specific features and datatypes. This is similar to Oracle Provider for OLE DB, where ActiveX Data Objects (ADO) provides an automation layer that exposes a programming model. ADO.NET provides a similar programming model, but without the automation layer for better performance. ODP.NET uses Oracle native APIs to access Oracle data and features from any .NET application.

See Also: *Oracle Database Express Edition 2 Day Plus .NET Developer Guide* for information about application development with Oracle Database XE in Microsoft .NET

Oracle Database Extensions for .NET

Oracle Database Extensions for .NET provides the following:

- A Common Language Runtime (CLR) host for Oracle Database XE
- Data access through Oracle Data Provider for .NET classes
- Oracle Deployment Wizard for Visual Studio .NET

Oracle Database XE hosts Microsoft Common Language Runtime (CLR) in an external process, outside of the Oracle database process, but on the same computer. The integration of Oracle Database XE with the Microsoft Common Language Runtime (CLR) enables applications to run .NET stored procedures or functions on Oracle Database XE, Microsoft Windows 2003, Windows 2000, and Windows XP.

Stored procedures and functions can be written using any .NET compliant language, such as C# and VB.NET. These .NET stored procedures can be used in the same manner as other PL/SQL or Java stored procedures, and can be called from PL/SQL packages, procedures, functions, and triggers.

.NET procedures or functions are built into a .NET assembly, typically using Microsoft Visual Studio .NET. Oracle Data Provider for .NET is used in .NET stored procedures and functions to access data. After building .NET procedures and functions into a

.NET assembly, they can be deployed in Oracle Database XE, using the Oracle Deployment Wizard for .NET, a component of the Oracle Developer Tools for Visual Studio .NET.

See Also: *Oracle Database Express Edition 2 Day Plus .NET Developer Guide* for information about application development with Oracle Database XE in Microsoft .NET

Oracle Developer Tools for Visual Studio .NET

Oracle Developer Tools is an add-on to Visual Studio .NET that provides graphical user interface (GUI) access to Oracle functionality.

Oracle Developer Tools include Oracle Explorer to browse your Oracle schema, designers and wizards to create and alter schema objects, and the ability to drag and drop schema objects onto your .NET form to automatically generate code. There is also a PL/SQL editor with integrated context-sensitive online Help. With Oracle Data Window, you can perform routine database tasks, such as inserting and updating Oracle data or testing stored procedures in the Visual Studio environment. For maximum flexibility, there is also a SQL Query Window for executing any SQL statement or SQL script.

See Also: *Oracle Database Express Edition 2 Day Plus .NET Developer Guide* for information about application development with Oracle Database XE in Microsoft .NET

Oracle Application Express

Oracle Application Express is a Web-based application development and deployment tool integrated with Oracle Database Express Edition. Oracle Application Express enables users with only a Web browser and limited programming experience to quickly create secure and scalable Web applications that can be instantly deployed to tens, hundreds, or thousands of users. The Application Builder tool assembles an HTML interface (or application) on top of database objects such as tables and procedures. Each application is a collection of pages linked together using tabs, buttons, or hypertext links.

See Also:

- *Oracle Database Express Edition 2 Day Plus Application Express Developer Guide* for tutorials with step-by-step instructions that explain how to create a variety of application components and complete applications using Oracle Application Express
- *Oracle Database Express Edition Application Express User's Guide* for information about building database-centric Web applications using the Oracle Application Express

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 XE. For more information about the JDBC API see:

`http://java.sun.com/products/jdbc`

Oracle Database JDBC brings provides features, such as complete support for JDBC 3.0 standard, complete support for JDBC RowSet (JSR-114), Advanced Connection

Caching (non-XA and XA connections), exposing SQL and PL/SQL data types to Java, and faster SQL data access. For information about the new JDBC features, see:

`http://www.oracle.com/technology/tech/java/sqlj_jdbc/pdf/twp_appdev_java_whats_new_4_java_jdbc_web_services.pdf`

For more information about the Oracle JDBC Drivers, see:

`http://www.oracle.com/technology/tech/java/sqlj_jdbc/index.html`

For an example of the use of PL/SQL with JDBC, see Appendix D, "Using a PL/SQL Procedure With JDBC".

> **See Also:** *Oracle Database Express Edition 2 Day Plus Java Developer Guide* for information about using Java to access and modify data in Oracle Database XE

PHP

PHP is a recursive acronym for PHP Hypertext Preprocessor. It is a widely-used, open-source, interpretive, HTML-centric, server-side scripting language. PHP is especially suited for Web development and can be embedded into HTML pages. PHP is comparable to languages such as Java Server Pages (JSP) and Oracle PL/SQL Server Pages (PSP). Zend Core for Oracle, developed in partnership with Zend Technologies, enables application development using PHP with Oracle Database XE.

For an example of the use of PL/SQL with PHP, see Appendix C, "Using a PL/SQL Procedure With PHP".

> **See Also:**
>
> - *Oracle Database Express Edition 2 Day Plus PHP Developer Guide* for information about application development using Zend Core for Oracle and Oracle Database XE
> - PHP Development Center at
>
>   `http://www.oracle.com/technology/tech/php/index.html`

2 Managing Database Objects

This section discusses creating and managing database objects in your schema, plus design considerations when developing applications with the Oracle Database Express Edition.

This section contains the following topics:

See Also: *Oracle Database SQL Reference* for information about schema objects, object names, and data types

Overview of Managing Objects

You need to create tables, indexes, and possibly other database objects in a schema before you start developing your application. A schema is a collection of database objects. A schema is owned by a database user and has the same name as that user, such as the `HR` schema. Schema objects are logical structures created by users. Objects can define areas of the database to hold data, such as tables, or can consist of just a definition, such as views.

Tables are the basic database objects and contain all the user data. When creating a table, it is important that you define that data that you want to store in the table. You need to specify the datatype of the data and any restrictions on the range of values. See "Using Datatypes" on page 2-7 and "Ensuring Data Integrity in Tables With Constraints" on page 2-13.

This chapter discusses tables, indexes, views, sequences, and synonyms. Other database (schema) objects include functions, packages, procedures, and triggers. Functions, packages, and procedures are discussed in Chapter 5, "Using Procedures, Functions, and Packages". Triggers are discussed in Chapter 6, "Using Triggers".

You can create, view, and manipulate database objects in your schema with Object Browser or SQL. With Object Browser, the underlying SQL is generated for you. In this chapter, the examples use Object Browser.

This section contains the following topics:

- Database Objects for Your Application on page 2-2
- Managing Database Objects With Object Browser on page 2-2
- Viewing Data in Tables With Object Browser on page 2-5
- Viewing Information With Object Reports on page 2-6

See Also:

- *Oracle Database Express Edition Application Express User's Guide* for a detailed description of the use of Object Browser to manage database objects
- *Oracle Database Express Edition 2 Day DBA* for information about getting started with Oracle Database Express Edition

Database Objects for Your Application

Some object types have many more management options than others, but most have a number of similarities. Every object in the database belongs to just one schema and has a unique name within that schema. Therefore, when you create an object, you must ensure it is in the schema where you intend to store it. Generally, you place all of the objects that belong to a single application in the same schema.

A database object name must abide by certain rules. For example, object names cannot be longer than 30 bytes and must begin with a letter. If you attempt to create an object with a name that violates any of these rules, then Oracle Database XE raises an error.

The following sections describe how to view, create, and manage the various types of objects in your database schemas.

Managing Database Objects With Object Browser

You can use the Object Browser page to create, modify, or view all your database objects. For example, with Object Browser you can create a table and then modify it by adding and deleting columns or adding constraints. You can also view all the objects that are currently used in a schema, such as those associated with the `HR` user.

To access the Object Browse page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To view the database objects or run the examples discussed in this guide, enter `HR` in the **Username** field and your password for the `HR` user account in the **Password** field. See "Sample HR Account" on page 1-5.
2. On the Database Home Page, click the **Object Browser** icon.

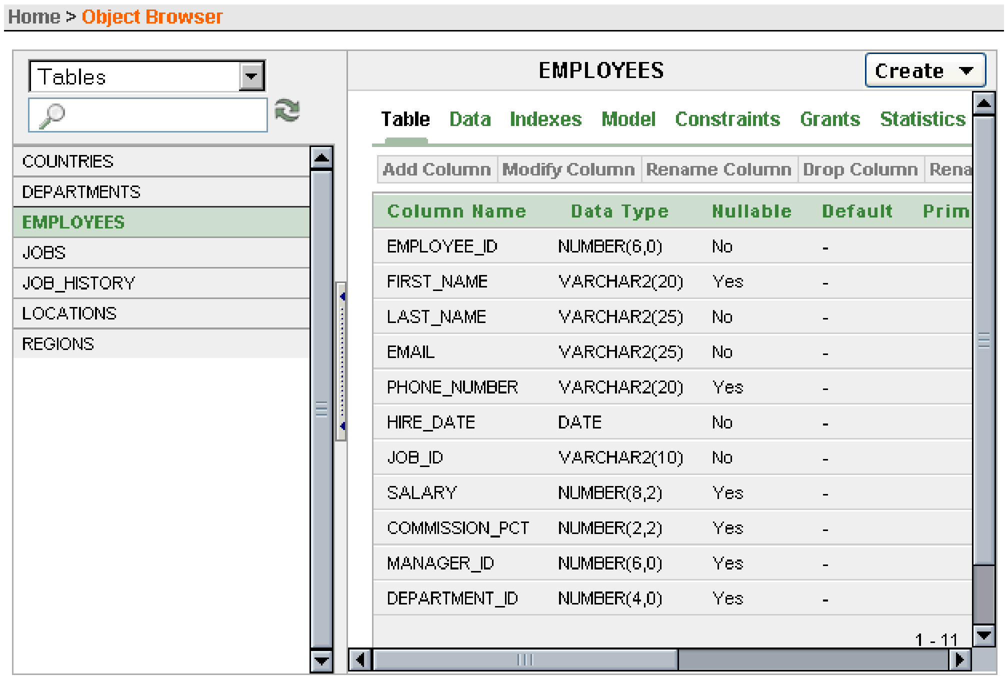
   The Object Browser page displays with two sections:

   - The Object Selection pane displays on the left side of the Object Browser page and lists database objects of a selected type within the current schema. For example, for the HR user the Tables object list includes `countries`, `departments`, `employees`, `jobs`, `job_history`, `locations`, and `regions`.
   - The Detail pane displays to the right of the page and displays detailed information about the selected object in the object list. You can click the tabs at the top of the Detail pane to view additional details about the current object.
3. On the Object Browser page, you can create, alter, and view database objects. Click the `HR EMPLOYEES` table in the Tables object list to display information about the structure of that table.

The information for the `employees` table includes the following about the columns and their datatypes:

```
Column Name                Data Type         Nullable
-------------------------- ----------------- ---------
EMPLOYEE_ID                NUMBER(6,0)       No
FIRST_NAME                 VARCHAR2(20)      Yes
LAST_NAME                  VARCHAR2(25)      No
EMAIL                      VARCHAR2(25)      No
PHONE_NUMBER               VARCHAR2(20)      Yes
HIRE_DATE                  DATE              No
JOB_ID                     VARCHAR2(10)      No
SALARY                     NUMBER(8,2)       Yes
COMMISSION_PCT             NUMBER(2,2)       Yes
MANAGER_ID                 NUMBER(6,0)       Yes
DEPARTMENT_ID              NUMBER(4,0)       Yes
```

Note the `employees` table uses numeric (`NUMBER`), character (`VARCHAR2`), and date (`DATE`) datatypes. See "Using Datatypes" on page 2-7.



Column Name	Data Type	Nullable	Default	Prim
EMPLOYEE_ID	NUMBER(6,0)	No	-	
FIRST_NAME	VARCHAR2(20)	Yes	-	
LAST_NAME	VARCHAR2(25)	No	-	
EMAIL	VARCHAR2(25)	No	-	
PHONE_NUMBER	VARCHAR2(20)	Yes	-	
HIRE_DATE	DATE	No	-	
JOB_ID	VARCHAR2(10)	No	-	
SALARY	NUMBER(8,2)	Yes	-	
COMMISSION_PCT	NUMBER(2,2)	Yes	-	
MANAGER_ID	NUMBER(6,0)	Yes	-	
DEPARTMENT_ID	NUMBER(4,0)	Yes	-	

1 - 11

4. In the Tables object list for the `HR` user, click the `DEPARTMENTS` table to view information about the structure of that table.

   The information for the `departments` table includes the following:

```
Column Name                Data Type         Nullable
-------------------------- ----------------- ---------
DEPARTMENT_ID              NUMBER(4,0)       No
DEPARTMENT_NAME            VARCHAR2(30)      No
MANAGER_ID                 NUMBER(6,0)       Yes
LOCATION_ID                NUMBER(4,0)       Yes
```

5. In the Tables object list for the `HR` user, click the `JOBS` table to view information about the structure of that table.

The information for the `jobs` table includes the following:

```
Column Name               Data Type        Nullable
------------------------- ---------------- ---------
JOB_ID                    VARCHAR2(10)     No
JOB_TITLE                 VARCHAR2(35)     No
MIN_SALARY                NUMBER(6,0)      Yes
MAX_SALARY                NUMBER(6,0)      Yes
```

6. In the Tables object list for the `HR` user, click the `JOB_HISTORY` table to view information about the structure of that table.

   The information for the `job_history` table includes the following:

```
Column Name               Data Type        Nullable
------------------------- ---------------- ---------
EMPLOYEE_ID               NUMBER(6,0)      No
START_DATE                DATE             No
END_DATE                  DATE             No
JOB_ID                    VARCHAR2(10)     No
DEPARTMENT_ID             NUMBER(4,0)      Yes
```

7. In the Tables object list for the `HR` user, click the `LOCATIONS` table to view information about the structure of that table.

   The information for the `locations` table includes the following:

```
Column Name               Data Type        Nullable
------------------------- ---------------- ---------
LOCATION_ID               NUMBER(4,0)      No
STREET_ADDRESS            VARCHAR2(40)     Yes
POSTAL_CODE               VARCHAR2(12)     Yes
CITY                      VARCHAR2(30)     No
STATE_PROVINCE            VARCHAR2(25)     Yes
COUNTRY_ID                CHAR(2)          Yes
```

   Note the use of the `CHAR` datatype for a fixed-length character field. See "What Are the Character Datatypes?" on page 2-8.

8. In the Tables object list for the `HR` user, click the `COUNTRIES` table to view information about the structure of that table.

   The information for the `countries` table includes the following:

```
Column Name               Data Type        Nullable
------------------------- ---------------- ---------
COUNTRY_ID                CHAR(2)          No
COUNTRY_NAME              VARCHAR2(40)     Yes
REGION_ID                 NUMBER           Yes
```

9. In the Tables object list for the `HR` user, click the `REGIONS` table to view information about the structure of that table.

   The information for the `regions` table includes the following:

```
Column Name               Data Type        Nullable
------------------------- ---------------- ---------
REGION_ID                 NUMBER           No
REGION_NAME               VARCHAR2(25)     Yes
```

10. In the object list, select **Views**, then click the `emp_details_view` view to display information about the structure of that view. That is contains columns from the `employees`, `departments`, `jobs`, `locations`, `countries`, and `regions` tables.

    The information for the `emp_details_view` view includes the following:

    ```
    Column Name              Data Type        Nullable
    ------------------------ ---------------- ---------
    EMPLOYEE_ID              NUMBER(6,0)      No
    JOB_ID                   VARCHAR2(10)     No
    MANAGER_ID               NUMBER(6,0)      Yes
    DEPARTMENT_ID            NUMBER(4,0)      Yes
    LOCATION_ID              NUMBER(4,0)      Yes
    COUNTRY_ID               CHAR(2)          Yes
    FIRST_NAME               VARCHAR2(20)     Yes
    LAST_NAME                VARCHAR2(25)     No
    SALARY                   NUMBER(8,2)      Yes
    COMMISSION_PCT           NUMBER(2,2)      Yes
    DEPARTMENT_NAME          VARCHAR2(30)     No
    JOB_TITLE                VARCHAR2(35)     No
    CITY                     VARCHAR2(30)     No
    STATE_PROVINCE           VARCHAR2(25)     Yes
    COUNTRY_NAME             VARCHAR2(40)     Yes
    REGION_NAME              VARCHAR2(25)     Yes
    ```

11. In the Object list, select other object types to display any existing objects of that type in the `HR` schema.

Viewing Data in Tables With Object Browser

In addition to viewing table names and table definitions, you can view the data stored in the table as well as the SQL statement used to display the data. You can also change the SQL statement to alter the result set.

To view table data:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `employees` table.
4. Click **Data** to display the rows of data in the tables.

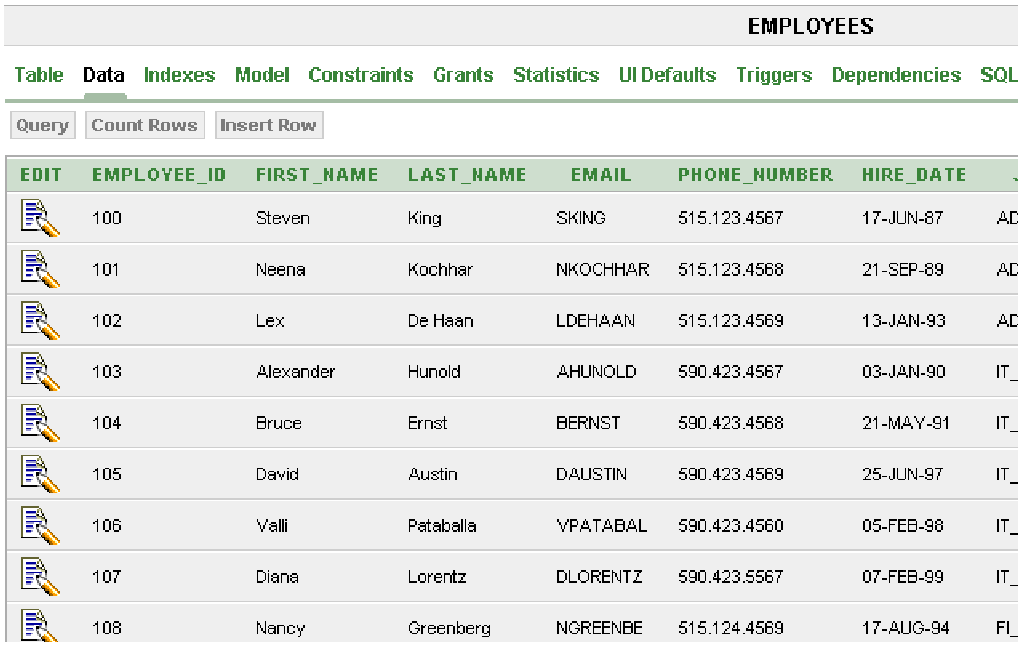

EMPLOYEES

Table Data Indexes Model Constraints Grants Statistics UI Defaults Triggers Dependencies SQL

Query Count Rows Insert Row

EDIT	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	
	100	Steven	King	SKING	515.123.4567	17-JUN-87	AD
	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD
	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD
	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_
	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_
	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_
	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_
	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_
	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_

You can also write your own SQL query using a `SELECT` statement to see the contents of a table. See "Running SQL Statements" on page 3-2.

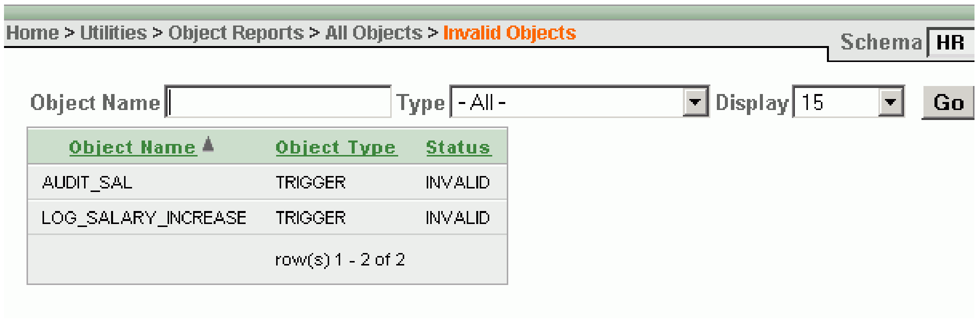
Viewing Information With Object Reports

You can run reports on database objects with the Reports feature of the Utilities tool. For example, you might want to run a report on all tables in the database, on all the columns in a specific table, or all database objects that are currently invalid.

To run a report on all invalid database objects:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Utilities** icon.

   The Utilities home page appears.
3. On the Utilities page, click **Object Reports**.
4. On the Object Reports page, click the **All Objects** icon.
5. On the All Objects page, click the **Invalid Objects** icon.
6. On the Invalid Objects page, select `-All-` from **Type** list.
7. Click the **Go** button to display a report on all invalid objects in the database.


Home > Utilities > Object Reports > All Objects > Invalid Objects

Schema HR

Object Name Type - All - Display 15 Go

Object Name	Object Type	Status
AUDIT_SAL	TRIGGER	INVALID
LOG_SALARY_INCREASE	TRIGGER	INVALID

row(s) 1 - 2 of 2

See Also: *Oracle Database Express Edition Application Express User's Guide* for detailed information about using Object Reports

Using Datatypes

A datatype associates a fixed set of properties with values that are used in a column of a table or in an argument of a procedure or function. The properties of datatypes cause Oracle Database XE to treat values of one datatype differently from values of another datatype. For example, Oracle Database XE can use the addition operator on values of numeric datatypes, but not with values of some other datatypes.

The datatypes supported by Oracle Database Express Edition include:

- Character datatypes
- Numeric datatypes
- Date and time (date-time) datatypes
- Large Object (LOB) datatypes

When you create a table, you must specify a datatype for each of its columns to define the nature of the data to be stored in the column. For example, a column defined as a DATE datatype cannot accept the value `February 29` (except for a leap year) or the values `2` or `SHOE`. When specifying a datatype, you can also indicate the longest value that can be placed in the column. In most cases, you only need columns of `NUMBER`, `VARCHAR2`, and `DATE` datatypes to create a definition of a table.

To view the datatypes specified for the columns in a database table, such as the `employees` table, you can use the Object Browser page. See "Managing Database Objects With Object Browser" on page 2-2. You can use also use the `DESCRIBE` command entered at SQL Command Line (SQL*Plus). For information about the SQL Command Line `DESCRIBE` command, see "SQL Command Line DESCRIBE Command" on page A-3.

This section contains the following topics:

- Storing Character Data on page 2-7
- Storing Numeric Data on page 2-9
- Storing Date and Time Data on page 2-10
- Storing Large Objects on page 2-12

See Also:

- *Oracle Database SQL Reference* for complete reference information about the SQL datatypes
- *Oracle Database SQL Reference* for a complete list of built-in datatypes in an Oracle database
- *Oracle Database Concepts* to learn about Oracle built-in datatypes

Storing Character Data

This section contains the following topics:

- What Are the Character Datatypes? on page 2-8
- Choosing Between the Character Datatypes on page 2-8

What Are the Character Datatypes?

You can use the following SQL datatypes to store character (alphanumeric) data:

- The `VARCHAR2` datatype stores variable-length character literals.

  When creating a `VARCHAR2` column in a table, you must specify a string length between 1 and 4000 bytes for the `VARCHAR2` column. Set the size to the maximum number of characters to be stored in the column. For example, a column to hold the last name of employees can be restricted to 25 bytes by defining it as `VARCHAR2(25)`.

  For each row, Oracle Database XE stores each value in the column as a variable-length field unless a value exceeds the column's maximum length, in which case Oracle returns an error. Using `VARCHAR2` saves on space used by the table. For most cases where you need to store character data, you would use the `VARCHAR2` datatype.

- The `CHAR` datatype stores fixed-length character literals.

  When creating a `CHAR` column in a table, you must specify a string length between 1 and 2000 bytes for the `CHAR` column. For each row, Oracle Database XE stores each value in the column as a fixed-length field. If the value of the character data is less than specified length of the column, then the value is blank-padded to the fixed length. If a value is too large, Oracle Database XE returns an error.

- `NCHAR` and `NVARCHAR2` datatypes store only Unicode character data.

  The `NVARCHAR2` datatype stores variable-length Unicode character literals. The `NCHAR` datatype stores fixed-length Unicode character literals. See Chapter 7, "Working in a Global Environment" for information about using Unicode data and globalization support.

See Also:

- *Oracle Database SQL Reference* for information about character datatypes
- *Oracle Database Globalization Support Guide* for information about globalization support

Choosing Between the Character Datatypes

When deciding which datatype to use for a column that will store character data in a table, consider the following:

- Space usage

  To store data more efficiently, use the `VARCHAR2` datatype. The `CHAR` datatype adds blanks to maintain a fixed column length for all column values, whereas the `VARCHAR2` datatype does not add extra blanks.

- Comparison semantics

  Use the `CHAR` datatype when trailing blanks are not important in string comparisons. Use the `VARCHAR2` datatype when trailing blanks are important in string comparisons.

- Future compatibility

  The `CHAR` and `VARCHAR2` datatypes are fully supported.

See Also: *Oracle Database SQL Reference* for more information about comparison semantics for these datatypes

Storing Numeric Data

This section contains the following topics:

- What Are the Numeric Datatypes? on page 2-9
- Using the NUMBER Datatype on page 2-9
- Using Floating-Point Number Formats on page 2-10

What Are the Numeric Datatypes?

The following SQL datatypes store numeric data:

- `NUMBER`
- `BINARY_FLOAT`
- `BINARY_DOUBLE`

Use the `NUMBER` datatype to store integers and real numbers in a fixed-point or floating-point format. Numbers using this datatype are guaranteed to be portable among different Oracle database platforms. For nearly all cases where you need to store numeric data, you would use the `NUMBER` datatype. When defining numeric data, you can use the precision option to set the maximum number of digits in the number, and the scale option to define how many of the digits are to the right of the decimal point. For example, a field to hold the salary of an employee can be defined as `NUMBER(8,2)`, providing 6 digits for the primary unit of currency (dollars, pounds, marks, and so on) and two digits for the secondary unit (cents, pennies, pfennigs, and so on).

Oracle Database XE provides the numeric `BINARY_FLOAT` and `BINARY_DOUBLE` datatypes exclusively for floating-point numbers. They support all of the basic functionality provided by the `NUMBER` datatype. However, while the `NUMBER` datatype uses decimal precision, `BINARY_FLOAT` and `BINARY_DOUBLE` datatypes use binary precision. This enables faster arithmetic calculations and usually reduces storage requirements.

See Also:

- *Oracle Database Concepts* for information about the internal format for the `NUMBER` datatype
- *Oracle Database SQL Reference* for more information about the `NUMBER`, `BINARY_FLOAT`, and `BINARY_DOUBLE` datatypes formats

Using the NUMBER Datatype

The `NUMBER` datatype stores zero as well as positive and negative fixed numbers with absolute values from 1.0 x 10^{-130} to (but not including) 1.0 x 10^{126}. If you specify an arithmetic expression whose value has an absolute value greater than or equal to 1.0 x 10^{126}, then Oracle Database XE returns an error.

The `NUMBER` datatype can be specified with a precision (*p*) and a scale (*s*) designator. Precision is the total number of significant decimal digits, where the most significant digit is the left-most, nonzero digit, and the least significant digit is the right-most, known digit. Scale is the number of digits from the decimal point to the least significant digit. The scale can range from -84 to 127. For examples, see Table 2–1 on page 2-10.

You can specify a `NUMBER` datatype as follows:

- `NUMBER(p)` for an integer

  This represents a fixed-point number with precision *p* and scale 0, and is equivalent to `NUMBER(p,0)`.

- `NUMBER(p, s)` for a fixed-point number

  This explicitly specifies the precision (*p*) and scale (*s*). It is good practice to specify the scale and precision of a fixed-point number column for extra integrity checking on input. Specifying scale and precision does not force all values to a fixed length. If a value exceeds the precision, then Oracle Database XE returns an error. If a value exceeds the scale, then Oracle Database XE rounds it.

- `NUMBER` for a floating-point number

  The absence of precision and scale designators specifies the maximum range and precision for an Oracle number.

Table 2–1 show how Oracle Database XE stores data using different values for precision and scale. Note that the values are rounded to the specified scale.

Table 2–1 Storage of Scale and Precision

Actual Data	Specified As	Stored As
123.8915	`NUMBER`	123.8915
123.8915	`NUMBER(3)`	124
123.8915	`NUMBER(4,1)`	123.9
123.8915	`NUMBER(5,2)`	123.89
123.8915	`NUMBER(6,3)`	123.892
123.8915	`NUMBER(7,4)`	123.8915
1.238915e2	`NUMBER(7,4)`	123.8915

Using Floating-Point Number Formats

The `BINARY_FLOAT` and `BINARY_DOUBLE` datatypes store floating-point data in the 32-bit IEEE 754 format and the double precision 64-bit IEEE 754 format respectively. Compared to the Oracle `NUMBER` datatype, arithmetic operations on floating-point data are usually faster for `BINARY_FLOAT` and `BINARY_DOUBLE`. High-precision values require less space when stored as `BINARY_FLOAT` and `BINARY_DOUBLE` datatypes.

The `BINARY_FLOAT` datatype has a maximum positive value equal to `3.40282E+38F` and the minimum positive value equal to `1.17549E-38F`.

The `BINARY_DOUBLE` datatype has a maximum positive value equal to `1.79769313486231E+308` and the minimum positive value equal to `2.22507485850720E-308`.

Storing Date and Time Data

Oracle Database XE stores dates in its own internal format that corresponds to century, year, month, day, hour, minute, and second. For input and output of dates, the standard Oracle Database XE default date format is `DD-MON-RR`. The `RR` date-time format element enables you store 20th century dates in the 21st century by specifying only the last two digits of the year. Time is stored in a 24-hour format as `HH24:MI:SS`.

Oracle Database Express Edition provides various SQL functions to calculate and convert date-time data. For examples, see "Using Date Functions" on page 3-15 and

"Using Conversion Functions" on page 3-16. For more information about manipulating date formats on a global level, see Chapter 7, "Working in a Global Environment".

This section contains the following topic:

- Using DATE and TIMESTAMP Datatypes on page 2-11

> **See Also:**
>
> - *Oracle Database SQL Reference* for more information about date and time formats
> - *Oracle Database Concepts* for information about Julian dates

Using DATE and TIMESTAMP Datatypes

Oracle Database supports the following date and time (date-time) datatypes:

- `DATE`
- `TIMESTAMP`
- `TIMESTAMP WITH TIME ZONE`
- `TIMESTAMP WITH LOCAL TIME ZONE`

Table 2–2 shows examples of `DATE` and `TIMESTAMP` datatypes.

Table 2–2 DATE and TIMESTAMP Examples

Datatype	Example
`DATE`	09-DEC-05
`TIMESTAMP`	09-DEC-05 02.05.49.000000 PM
`TIMESTAMP WITH TIME ZONE`	09-DEC-05 02.05.49.000000 PM -08:00
`TIMESTAMP WITH LOCAL TIME ZONE`	09-DEC-05 02.05.49.000000 PM

> **See Also:** *Oracle Database SQL Reference* for information about `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE` datatypes

Using the DATE Datatype Use the `DATE` datatype to store point-in-time values (dates and times) in a table. For example, a column to hold the date that an employee is hired can by defined as a DATE datatype. An application that specifies the time for a job might also use the `DATE` datatype. For most cases where you need to store date data, you would use the `DATE` datatype.

`DATE` columns are automatically formatted by Oracle Database XE to include a date and time component. The `DATE` datatype stores the century, year, month, day, hours, minutes, and seconds. The valid date range is from January 1, 4712 BC to December 31, 9999 AD. Although both the date and time are stored in a date column, by default, the date portion is automatically displayed for you, when retrieving date data. However, Oracle Database Express Edition enables you great flexibility in how you can display your dates and times. See "Using Date Functions" on page 3-15.

Using the TIMESTAMP Datatype Use the `TIMESTAMP` datatype to store values that are precise to fractional seconds. An application that must decide which of two events occurred first might use `TIMESTAMP`.

Using the TIMESTAMP WITH TIME ZONE Datatype Because the `TIMESTAMP WITH TIME ZONE` datatype can also store time zone information, it is particularly suited for recording date information that must be gathered or coordinated across geographic regions.

Using the TIMESTAMP WITH LOCAL TIME ZONE Datatype Use the `TIMESTAMP WITH LOCAL TIME ZONE` datatype when the time zone is not significant. For example, you might use it in an application that schedules teleconferences, where participants each see the start and end times for their own time zone.

The `TIMESTAMP WITH LOCAL TIME ZONE` datatype is appropriate for two-tier applications in which you want to display dates and times that use the time zone of the client system. It is generally inappropriate in three-tier applications because data displayed in a Web browser is formatted according to the time zone of the Web server, not the time zone of the browser. The Web server is the database client, so its local time is used.

Storing Large Objects

Large Objects (LOBs) are a set of datatypes that are designed to hold large amounts of data. A LOB can hold up to a maximum size ranging from 8 terabytes to 128 terabytes depending on how your database is configured. Storing data in LOBs enables you to access and manipulate the data efficiently in your application.

The `BLOB`, `CLOB`, and `NCLOB` datatypes are internal LOB datatypes and are stored in the database. The `BFILE` datatype is the only external LOB datatype and is stored in an operating system file, outside the database.

> **See Also:** *Oracle Database Application Developer's Guide - Large Objects* for more information about Large Object datatypes

Managing Tables

Tables are the basic unit of data storage in an Oracle database. They hold all user-accessible data. A table is a two-dimensional object made up of columns and rows. For example, the `employees` table includes (vertical) columns called `employee_id`, `first_name`, and `last_name`. Each (horizontal) row in the table contains a value for employee name and ID number. The most common type of table in an Oracle database is a relational table.

This section contains the following topics:

- Ensuring Data Integrity in Tables With Constraints on page 2-13
- Creating a Table on page 2-16
- Adding a Column To a Table on page 2-18
- Modifying a Column In a Table on page 2-18
- Dropping a Column From a Table on page 2-19
- Adding a Check Constraint on page 2-19
- Adding a Unique Constraint on page 2-20
- Adding a Primary Key Constraint on page 2-21
- Adding a Foreign Key Constraint on page 2-22
- Viewing Existing Constraints on page 2-23

- Disabling and Enabling a Constraint on page 2-23
- Dropping a Constraint on page 2-24
- Adding Data to a Table on page 2-5
- Modifying Data in a Table on page 2-25
- Removing a Row in a Table on page 2-26
- Dropping a Table on page 2-26

See Also:

- *Oracle Database Administrator's Guide* for information about managing tables
- *Oracle Database Concepts* for conceptual information about tables types
- *Oracle Database SQL Reference* for the syntax required to create and alter tables
- *Oracle Database Express Edition Application Express User's Guide* for information about managing tables

Ensuring Data Integrity in Tables With Constraints

With Oracle Database XE, you can define integrity constraints to enforce business rules on data in your tables to preserve the integrity of the data. Business rules specify conditions and relationships that must always be true, or must always be false. For example, in a table containing employee data, the employee e-mail column must be unique. Similarly, in this table you cannot have two employees with the same employee ID.

When an integrity constraint applies to a table, all data in the table must conform to the corresponding rule. When you issue a SQL statement that inserts or modifies data in the table, Oracle Database XE ensures that the new data satisfies the integrity constraint, without the need to do any checking within your program. Any attempt to insert, update, or remove a row that violates a constraint results in an error, and the statement is rolled back. Likewise, any attempt to apply a new constraint to a populated table also results in an error if any existing row violates the new constraint.

Constraints can be created and, in most cases, modified with a number of different status values. The options include enabled or disabled, which determine if the constraint is checked when rows are added, modified, or removed; and deferred or immediate, which cause constraint validation to occur at the end of a transaction or at the end of a statement, respectively.

You can enforce rules by defining integrity constraints more reliably than by adding logic to your application. Oracle Database XE can check that all the data in a table obeys an integrity constraint faster than an application can.

Constraints can be defined at the column level or at the table level:

- Column-level constraints are syntactically defined where the column to which the constraint applies is defined. These constraints determine what values are valid in the column. When creating a table with Object Browser, the only constraint defined at the column level is the `NOT NULL` constraint, which requires that a value is included in this column for every row in the table.

- Table-level constraints are syntactically defined at the end of the table definition and apply to the entire table. With Object Browser, you can create primary key, foreign key, unique, and check constraints.

This section contains the following topics:

See Also: *Oracle Database Concepts* for more information about constraints

Column Default Value

You can define default values that are values that are automatically stored in the column whenever a new row is inserted without a value being provided for the column. When you define a column with a default value, any new rows inserted into the table store the default value unless the row contains an alternate value for the column. Assign default values to columns that contain a typical value. For example, in the `employees` table, if most employees work in the sales department, then the default value for the `department_id` column can be set to the ID of the sales department.

Depending on your business rules, you might use default values to represent zero or `FALSE`, or leave the default values as `NULL` to signify an unknown value. Default values can be defined using any literal, or almost any expression including `SYSDATE`, which is a SQL function that returns the current date. For an example of the use of the `DEFAULT` column value, see Example 3–34 on page 3-23.

NOT NULL Constraint

The `NOT NULL` constraint is a column-level constraint that requires that the column must contain a value whenever a row is inserted or updated. The `NOT NULL` constraint must be defined as part of the column definition.

Use a `NOT NULL` constraint when the data is required for the integrity of the database. For example, if all employees must belong to a specific department, then the column that contains the department identifier should be defined with a `NOT NULL` constraint. On the other hand, do not define a column as `NOT NULL` if the data might be unknown or might not exist when rows are added or changed, for example, the second, optional line in a mailing address.

A primary key constraint automatically adds a `NOT NULL` constraint to the columns included in the primary key, in addition to enforcing uniqueness among the values.

For an example of the use of the `NOT NULL` constraint, see "Creating a Table" on page 2-16.

Check Constraint

A check constraint requires that a column (or combination of columns) satisfies a condition for every row in the table. A check constraint must be a Boolean expression that is evaluated using the column value about to be inserted or updated to the row.

Use check constraints when you need to enforce integrity rules based on logical expressions, such as comparisons. Never use CHECK constraints when any of the other types of integrity constraints can provide the necessary checking.

Examples of check constraints include the following:

- A check constraint on employee salaries so that no salary value is less than 0.
- A check constraint on department locations so that only the locations `Boston`, `New York`, and `Dallas` are allowed.
- A check constraint on the salary and commissions columns to prevent the commission from being larger than the salary.

For an example of the use of the check constraint, see "Adding a Check Constraint" on page 2-19.

Unique Constraint

A unique constraint requires that every value in a column be unique. That is, no two rows can have duplicate values in a specified column or combination of columns.

Choose columns for unique constraints carefully. The purpose of these constraints is different from that of primary keys. Unique key constraints are appropriate for any column where duplicate values are not allowed. Primary keys identify each row of the table uniquely, and typically contain values that have no significance other than being unique. In the `employees` table, the `email` column has a unique key constraint because it is important that the e-mail address for each employee is unique. Note that the `email` column has a `NOT NULL` constraint.

Some examples of good unique keys include:

- An employee social security number, where the primary key might be the employee number
- A truck license plate number, where the primary key might be the truck number
- A customer phone number, consisting of the two columns `area_code` and `local_phone`, where the primary key might be the customer number
- A department name and location, where the primary key might be the department number

For an example of the use of the unique constraint, see "Adding a Unique Constraint" on page 2-20.

Primary Key Constraint

A primary key requires that a column (or combination of columns) be the unique identifier of the row and ensures that no duplicate rows exist. A primary key column cannot contain NULL values. Each table can have only one primary key.

Use the following guidelines when selecting a primary key:

- Whenever practical, create a sequence number generator to generate unique numeric values for your primary key values. See "Managing Sequences" on page 2-34.
- Choose a column whose data values are unique, because the purpose of a primary key is to uniquely identify each row of the table.
- Choose a column whose data values are never changed. A primary key value is only used to identify a row in the table, and its data should never be used for any other purpose. Therefore, primary key values should rarely or never be changed.

- Choose a column that does not contain any null values. A `PRIMARY KEY` constraint, by definition, does not allow any row to contain a null value in any column that is part of the primary key.
- Choose a column that is short and numeric. Short primary keys are easy to type.
- Minimize your use of composite primary keys. A composite primary key constraint applies to more than one column. Although composite primary keys are allowed, they do not satisfy all of the other recommendations. For example, composite primary key values are long and cannot be assigned by sequence numbers.

For an example of the use of the primary key constraint, see "Adding a Primary Key Constraint" on page 2-21.

Foreign Key Constraint

Whenever two tables contain one or more common columns, you can enforce the relationship between the tables through a referential integrity constraint with a foreign key. A foreign key requires that all column values in the child table exist in the parent table. The table that includes the foreign key is called the dependent or child table. The table that is referenced is called the parent table.

An example of a foreign key constraint is when the department column of the `employees` table (child) must contain a department ID that exists in the `departments` table (parent).

Foreign keys can be made up of multiple columns. Such a composite foreign key must reference a composite primary or unique key of the exact same structure, with the same number of columns and the same datatypes. Because composite primary and unique keys are limited to 32 columns, a composite foreign key is also limited to 32 columns. You must use the same datatype for corresponding columns in the parent and child tables. The column names do not need to match.

For performance purposes, you might want to add an index to the columns you define in a child table when adding a foreign key constraint. Oracle Database XE does not do this for you automatically. See "Indexes for Use with Constraints" on page 2-28 and "Creating an Index" on page 2-29.

When you create a foreign key constraint on a table, you can specify the action to take when rows are deleted in the referenced (parent) table. These actions include:

- Disallow Delete - Blocks the delete of rows from the referenced table when there are dependent rows in the table.
- Cascade Delete - Deletes the dependent rows from the table when the corresponding parent table row is deleted from the referenced table.
- Null on Delete - Sets the foreign key column values in the table to null values when the corresponding table row is deleted from the referenced table.

For an example of the use of the foreign key constraint, see "Adding a Foreign Key Constraint" on page 2-22.

Creating a Table

You can use the Object Browser page to create a table. The procedure in this section creates a table that contains personal information for employees in the `employees` sample table.

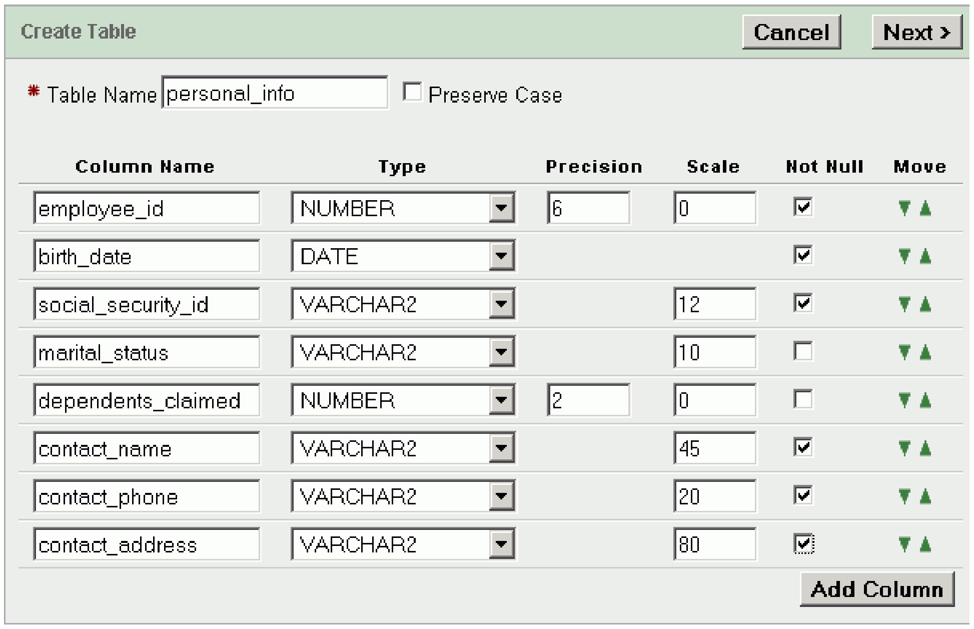
To create a table:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the object list under **Create**, select **Table**.
4. In the **Table Name** field, enter the name of the table (`personal_info`).
5. Enter the following column names and datatypes and enable the `NOT NULL` constraint where designated. The `NOT NULL` constraint specifies that a value must be entered in the column.

```
employee_id             NUMBER(6,0)   NOT NULL
birth_date              DATE          NOT NULL
social_security_id      VARCHAR2(12)  NOT NULL
marital status          VARCHAR2(10)
dependents_claimed      NUMBER(2,0)
contact_name            VARCHAR2(45)  NOT NULL
contact_phone           VARCHAR2(20)  NOT NULL
contact_address         VARCHAR2(80)  NOT NULL
```

   For information about datatypes, see "Using Datatypes" on page 2-7. For information about the `NOT NULL` constraint, see "NOT NULL Constraint" on page 2-14.

   Ensure that **Preserve Case** has been left unchecked so that names are stored in the default manner (uppercase), which avoids any extra overhead.



* Table Name: personal_info   ☐ Preserve Case

Column Name	Type	Precision	Scale	Not Null	Move
employee_id	NUMBER	6	0	✓	▼▲
birth_date	DATE			✓	▼▲
social_security_id	VARCHAR2		12	✓	▼▲
marital_status	VARCHAR2		10	☐	▼▲
dependents_claimed	NUMBER	2	0	☐	▼▲
contact_name	VARCHAR2		45	✓	▼▲
contact_phone	VARCHAR2		20	✓	▼▲
contact_address	VARCHAR2		80	✓	▼▲

6. After you have enter the column information, click **Next**.
7. On the Primary Key page, do not create a key at this time. Click the **Next** button. See "Adding a Primary Key Constraint" on page 2-21.
8. On the Foreign Key page, do not create a key at this time. Click the **Next** button. See "Adding a Foreign Key Constraint" on page 2-22.

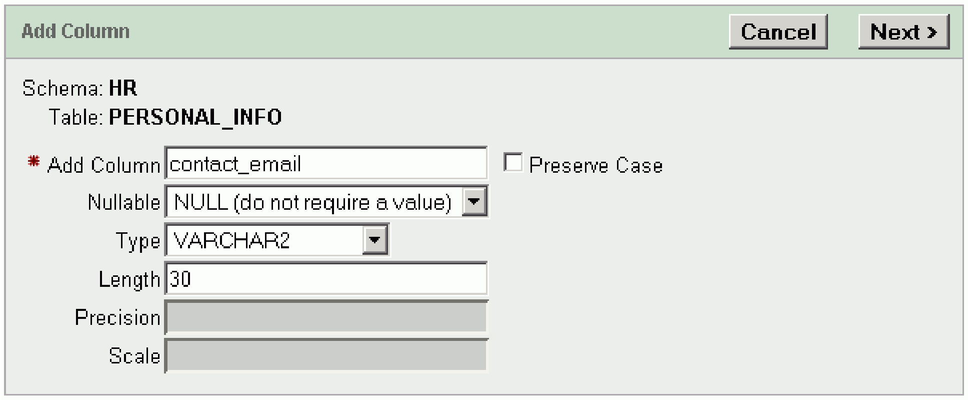
9. On the Constraints page, do not create a constraint at this time. Click the **Finish** button. See "Adding a Unique Constraint" on page 2-20 and "Adding a Check Constraint" on page 2-19.
10. On the Create Table page, click the **SQL** button to view the SQL statements that produce the table. This option shows the statement even if it is incomplete. You need to complete your input to see the complete SQL statement when using this option.
11. Click the **Create** button to create the table.

Adding a Column To a Table

You can use Object Browser to add columns to a table.

To add a column to a table:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** and then click the `personal_info` table that you previously created.
4. Click **Add Column**.
5. Enter the data to add a column named `contact_email`. The column can be `NULL`. The datatype is `VARCHAR2` with a length of `30`.



Schema: HR
Table: PERSONAL_INFO

* Add Column	contact_email ☐ Preserve Case
Nullable	NULL (do not require a value)
Type	VARCHAR2
Length	30
Precision	
Scale	

6. Click the **Next** button.
7. Click the **Finish** button to complete the action.

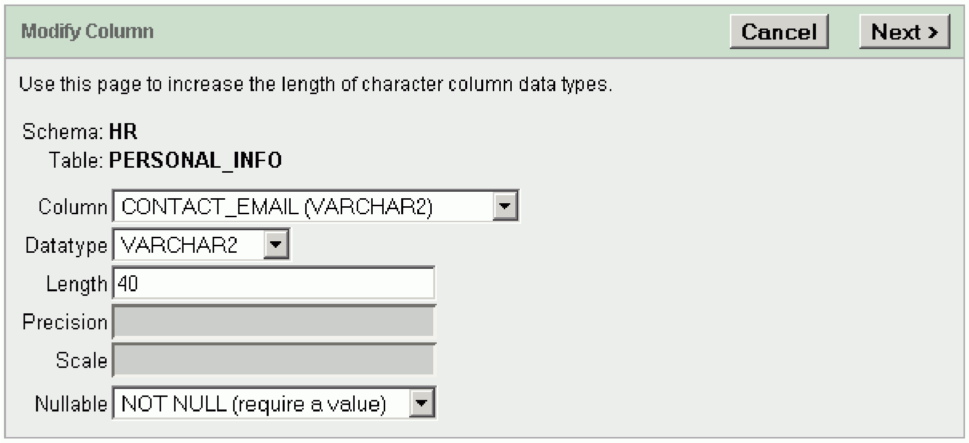
Modifying a Column In a Table

You can use Object Browser to modify a column in a table.

To modify a column in a table:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** and then click the `personal_info` table that you previously created.
4. Click **Modify Column**.

5. Select the `contact_email` column. Change the datatype to `VARCHAR2` with a length of `40`. Change the column to be `NOT NULL`.



Use this page to increase the length of character column data types.

Schema: **HR**
Table: **PERSONAL_INFO**

Column	CONTACT_EMAIL (VARCHAR2)
Datatype	VARCHAR2
Length	40
Precision	
Scale	
Nullable	NOT NULL (require a value)

6. Click the **Next** button.
7. Click the **Finish** button to complete the action.

Dropping a Column From a Table

You can use Object Browser to delete columns in a table. Before you do delete a column, make sure the data in that column is not going to be needed later.

To delete a column:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click **Drop Column**.
5. Select the `contact_address` column and click the **Next** button.
6. Click the **Finish** button to complete the action.

Adding a Check Constraint

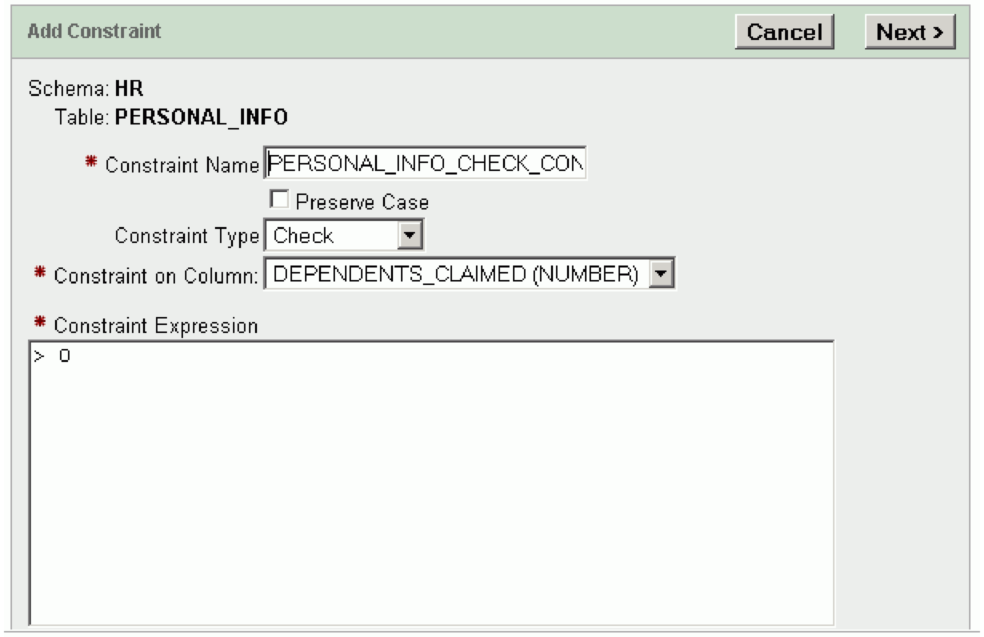
You can use Object Browser to add a constraint to a table after it has been created. In the `personal_info` table, you might want to check that the number of dependents claimed is always greater than `0`. For information about the check constraint, see "Check Constraint" on page 2-14.

To add a check constraint:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click the **Constraints** tab.
5. Click the **Create** button.

6. On the Add Constraint page, use the following information to complete the page.

   Constraint Name: `PERSONAL_INFO_CHECK_CON`
   Constraint Type: `Check`
   Constraint on Column: `DEPENDENTS_CLAIMED(NUMBER)`
   Constraint Expression: `> 0`



Schema: **HR**
Table: **PERSONAL_INFO**
* Constraint Name: PERSONAL_INFO_CHECK_CON
Constraint Type: Check
* Constraint on Column: DEPENDENTS_CLAIMED (NUMBER)
* Constraint Expression:

```
> 0
```

7. Click the **Next** button.
8. Click the **Finish** button to complete the action.

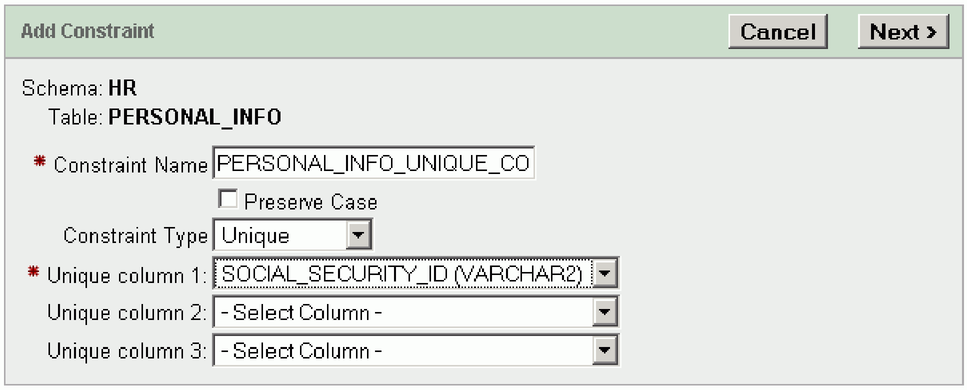
Adding a Unique Constraint

You can use Object Browser to add a constraint to a table after it has been created. In the `personal_info` table, you might want to enforce the rules so that each social security ID is unique. For information about the unique constraint, see "Unique Constraint" on page 2-15.

To add a unique constraint:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click the **Constraints** tab.
5. Click the **Create** button.
6. On the Add Constraint page, use the following information to complete the page.

   Constraint Name: `PERSONAL_INFO_UNIQUE_CON`
   Constraint Type: `Unique`
   Unique Column 1: `SOCIAL_SECURITY_ID(VARCHAR2)`



Schema: HR
Table: PERSONAL_INFO

* Constraint Name: PERSONAL_INFO_UNIQUE_CO
☐ Preserve Case
Constraint Type: Unique
* Unique column 1: SOCIAL_SECURITY_ID (VARCHAR2)
Unique column 2: - Select Column -
Unique column 3: - Select Column -

7. Click the **Next** button.
8. Click the **Finish** button to complete the action.

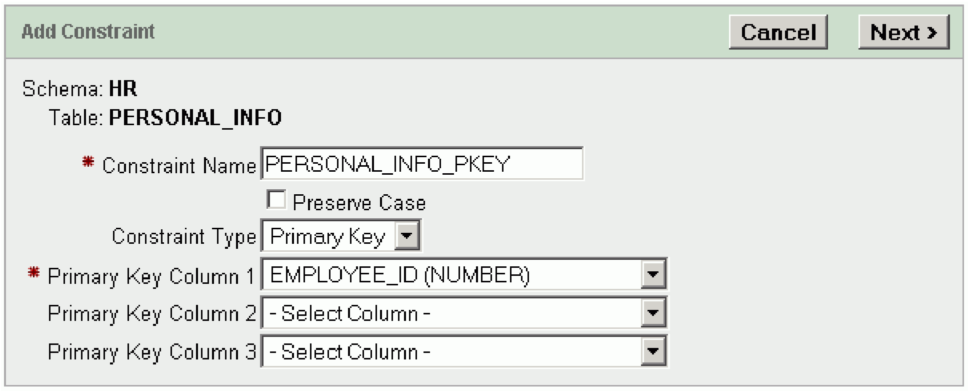
Adding a Primary Key Constraint

You can use Object Browser to add a primary key constraint on a column in a table. The primary key uniquely identifies each record (row) that is inserted in the table and ensures that no duplicate rows exist. For information about the primary key constraint, see "Primary Key Constraint" on page 2-15.

To add a primary key constraint:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click the **Constraints** tab.
5. Click the **Create** button.
6. On the Add Constraint page, use following information to complete the page:

   Constraint Name: `PERSONAL_INFO_PKEY`
   Constraint Type: `Primary Key`
   Primary Key Column 1: `EMPLOYEE_ID(NUMBER)`



Schema: HR
Table: PERSONAL_INFO

* Constraint Name: PERSONAL_INFO_PKEY
☐ Preserve Case
Constraint Type: Primary Key
* Primary Key Column 1: EMPLOYEE_ID (NUMBER)
Primary Key Column 2: - Select Column -
Primary Key Column 3: - Select Column -

7. Click the **Next** button.
8. Click the **Finish** button to complete the action.

Adding a Foreign Key Constraint

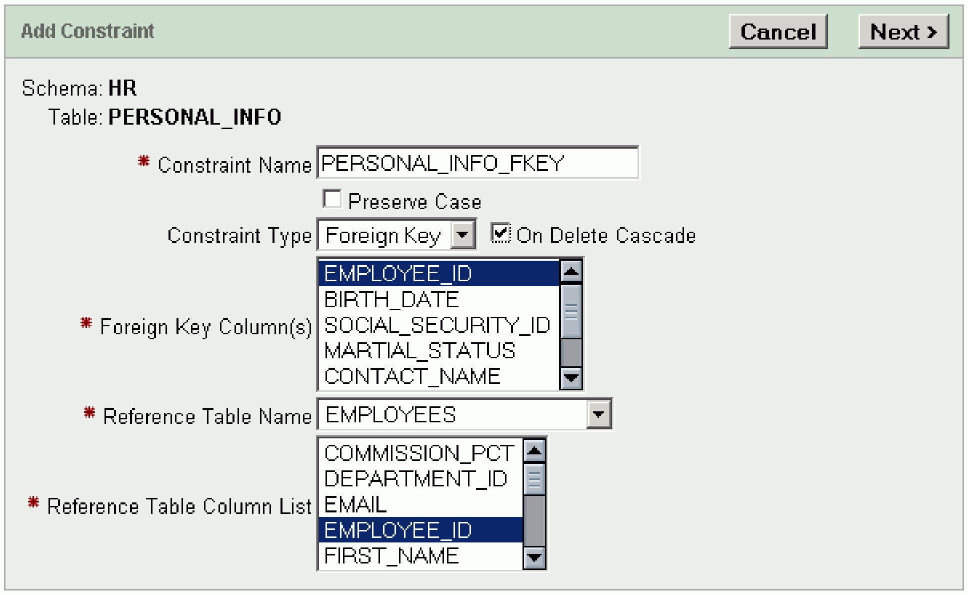
You can use Object Browser to add a foreign key constraint on a column in one table to a column in a reference table. This ensures that a value inserted in a column matches a valid value in the reference table. For information about the foreign key constraint, see "Foreign Key Constraint" on page 2-16.

To add a foreign key constraint:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click the **Constraints** tab.
5. Click the **Create** button.
6. On the Add Constraint page, select **Foreign Key** for the Constraint Type. Use following information to complete the page:

   Constraint Name: `PERSONAL_INFO_FKEY`
   Foreign Key Column: `EMPLOYEE_ID`
   Reference Table Name: `EMPLOYEES`
   Reference Table Column List: `EMPLOYEE_ID`

   Do not check the **Preserve Case** box. Check the **On Delete Cascade** box.



Schema: **HR**
Table: **PERSONAL_INFO**

* Constraint Name PERSONAL_INFO_FKEY
☐ Preserve Case
Constraint Type Foreign Key ☑ On Delete Cascade
* Foreign Key Column(s) EMPLOYEE_ID, BIRTH_DATE, SOCIAL_SECURITY_ID, MARTIAL_STATUS, CONTACT_NAME
* Reference Table Name EMPLOYEES
* Reference Table Column List COMMISSION_PCT, DEPARTMENT_ID, EMAIL, EMPLOYEE_ID, FIRST_NAME

7. Click the **Next** button.
8. Click the **Finish** button to complete the action.

Viewing Existing Constraints

You can use Object Browser to view existing constraints on a table.

To view constraints:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click the **Constraints** tab to view a list of the constraints on the `personal_info` table and information about the constraints.

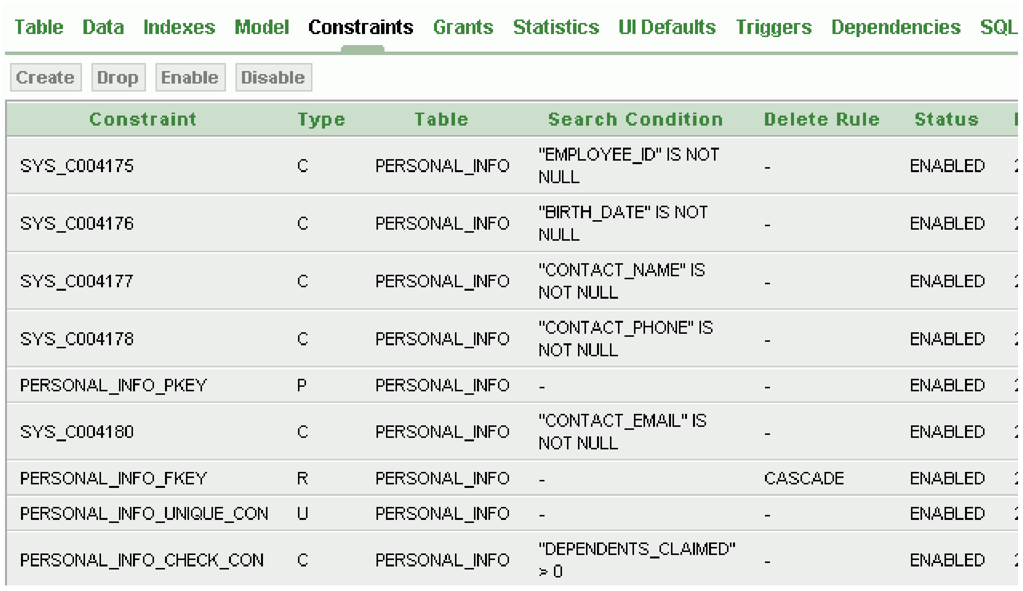


Table Data Indexes Model Constraints Grants Statistics UI Defaults Triggers Dependencies SQL
Create Drop Enable Disable

Constraint	Type	Table	Search Condition	Delete Rule	Status
SYS_C004175	C	PERSONAL_INFO	"EMPLOYEE_ID" IS NOT NULL	-	ENABLED
SYS_C004176	C	PERSONAL_INFO	"BIRTH_DATE" IS NOT NULL	-	ENABLED
SYS_C004177	C	PERSONAL_INFO	"CONTACT_NAME" IS NOT NULL	-	ENABLED
SYS_C004178	C	PERSONAL_INFO	"CONTACT_PHONE" IS NOT NULL	-	ENABLED
PERSONAL_INFO_PKEY	P	PERSONAL_INFO	-	-	ENABLED
SYS_C004180	C	PERSONAL_INFO	"CONTACT_EMAIL" IS NOT NULL	-	ENABLED
PERSONAL_INFO_FKEY	R	PERSONAL_INFO	-	CASCADE	ENABLED
PERSONAL_INFO_UNIQUE_CON	U	PERSONAL_INFO	-	-	ENABLED
PERSONAL_INFO_CHECK_CON	C	PERSONAL_INFO	"DEPENDENTS_CLAIMED" > 0	-	ENABLED

Disabling and Enabling a Constraint

You can use Object Browser to disable or enable a constraint.

To disable and enable a constraint:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click the **Constraints** tab.
5. Click the **Disable** button.
6. On the Disable Constraint page, select the check constraint that you created on the `dependents_claimed` column (`PERSONAL_INFO_CHECK_CON`). See "Viewing Existing Constraints" on page 2-23 to determine the name of the constraint.
7. Click the **Next** button.
8. Click the **Enable** button on the Constraints tab.

9. On the Enable Constraint page, select the check constraint that you created on the `dependents_claimed` column (`PERSONAL_INFO_CHECK_CON`). See "Viewing Existing Constraints" on page 2-23 to determine the name of the constraint.
10. Click the **Next** button.
11. Click the **Finish** button to complete the action.

Dropping a Constraint

You can use Object Browser to drop constraints from a table. Although you do not have to disable a constraint before dropping it, you can determine whether the constraint can be dropped by attempting to disable it first. If a constraint in a parent table enforces a foreign key constraint in a child table, and if the child table contains dependent rows, then the constraint cannot always be disabled or dropped.

Continuing with the current example, you drop the check constraint that you created earlier in the section, "Adding a Check Constraint" on page 2-19.

To drop a constraint:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click the **Constraints** tab.
5. Click the **Drop** button.
6. On the Drop Constraint page, select the check constraint that was created on the `contact_email` column that specifies a `NOT NULL` constraint. See "Viewing Existing Constraints" on page 2-23 to determine the name of the constraint, such as `SYS_C004180`.
7. Click the **Next** button.
8. Click the **Finish** button to complete the action.

Adding Data to a Table

You can add (or insert) a row of data to a table with Object Browser.

To add data to a table:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click **Data**.
5. Click the **Insert Row** button.
6. On the Create Row page, enter the following values:

   employee_id: `142`
   birth_date: `01-SEP-65`
   social_security_id: `555-11-4444`

marital_status: `Married`
dependents_claimed: `4`
contact_name: `Marilyn Davies`
contact_phone: `15552229999`
contact_email: `marilyn.davies@mycompany.com`

Note that when you add the data to the `personal_info` table, the values must conform to any constraints on the table. For example, the `contact_email` value must be 40 characters or less and the `employee_id` value must match a value in the `employee_id` column of the `employees` table. If data is entered that violates any constraint, then an error displays when you attempt to create a row.

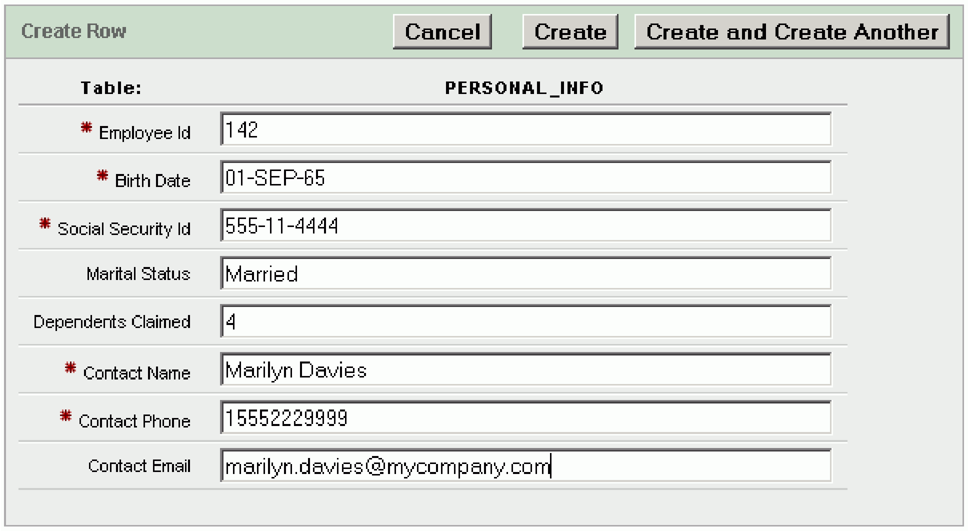


Table:	PERSONAL_INFO
* Employee Id	142
* Birth Date	01-SEP-65
* Social Security Id	555-11-4444
Marital Status	Married
Dependents Claimed	4
* Contact Name	Marilyn Davies
* Contact Phone	15552229999
Contact Email	marilyn.davies@mycompany.com

7. Click the **Create and Create Another** button to insert the row of data and create another row in the table.
8. In the Create Row page, enter the following values:

   employee_id: `143`
   birth_date: `01-MAR-72`
   social_security_id: `555-77-4444`
   marital_status: `Single`
   dependents_claimed: `1`
   contact_name: `Carolyn Matos`
   contact_phone: `15553338888`
   contact_email: `carolyn.matos@myinternet.com`

9. Click the **Create** button to insert the row of data.

Modifying Data in a Table

You can use the Object Browser page to modify data in a table.

To modify data:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.

4. Click **Data**.
5. Click **Edit** next to `employee_id` equal to 142.
6. Change the value of `phone_number` to `15551118888`.
7. Click the **Apply Changes** button.

Removing a Row in a Table

You can use Object Browser to remove a row from a table.

To remove a row:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click **Data**.
5. Click **Edit** next to `employee_id` equal to 143.
6. Click the **Delete** button to remove this row from the table.
7. Click **OK** to confirm the delete action.

Dropping a Table

If you no longer need a table or its contents, then you can drop the table using Object Browser. Be certain that you do not need the data in the table before you drop it. It may be difficult and time-consuming to retrieve the records, if they can be retrieved, after you execute the drop operation.

To test this procedure, follow the procedure in "Creating a Table" on page 2-16 to create a table.

To drop a table:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables** then click the `personal_info` table that you previously created.
4. Click **Drop**.
5. Click the **Finish** button to complete the action.

Managing Indexes

Indexes are optional structures associated with tables. You can create them to improve query performance. Just as the index in this book helps you to quickly locate specific information, an Oracle Database XE index provides a quick access path to table data. Before you add additional indexes, examine the performance of your database. You can then compare performance after the new indexes are added.

You can create indexes on one or more columns of a table. After an index is created, it is automatically maintained and used by Oracle Database XE. Changes to the structure of a table or data in a table, such as adding new rows, updating rows, or deleting rows, are automatically incorporated into all relevant indexes.

This section contains the following topics:

See Also: *Oracle Database Express Edition Application Express User's Guide* for information about managing indexes

Index Types

Indexes can be categorized in a number of ways. The primary ways are:

- Normal Index

  A standard, B-tree index contains an entry for each value in the index key along with an address to the row where the value is stored. A B-tree index is the default and most common type of index in an Oracle database.

- Text Index

  An index is used by Oracle Text for text searching, such as full-text retrieval over documents and Web pages.

- Single-Column and Concatenated Indexes

  You can create an index on one column, which is called a single-column index, or on multiple columns, which is called a concatenated index. Concatenated indexes are useful when all of the columns are likely to be included in the `WHERE` clause of frequently executed SQL statements.

  For concatenated indexes, define the columns used in the index carefully so that the column with the fewest duplicate values is named first, the column with the next fewest duplicate values is second, and so on. Columns with many duplicate values or many rows with null values should not be included or should be the last named columns in the index definition.

- Ascending and Descending Indexes

  The default search through an index is from the lowest to highest value where character data is sorted by ASCII values, numeric data is sorted from smallest to largest number, and date data is sorted from the earliest to the latest value. This default behavior is performed by indexes created as ascending indexes. You can reverse the search order of an index by creating the related index with the descending option.

- Column and Function-Based Indexes

  Typically, an index entry is based on the values found in the columns of the table. This is a column index. Alternatively, you can create a function-based index in which the indexed value is derived from the table data. For example, to find character data that can be in mixed case, you could use a function-based index to search for the values as if they were all in uppercase characters.

Indexes for Use with Constraints

All enabled unique and primary keys require corresponding indexes. Oracle Database XE automatically creates the indexes necessary to support data integrity defined with constraints when you add or enable those constraints. For example, a column with the constraint that its values be unique causes Oracle Database XE to create a unique key index.

Note the following:

- Constraints use existing indexes where possible, rather than creating new ones.
- Unique and primary keys can use non unique and unique indexes. In addition, they can use just the first few columns of non unique indexes.
- At most, one unique or primary key can use each non unique index.
- The column orders in the index and the constraint do not need to match.
- For performance purposes, you might want to add an index to the columns you define in a child table when adding a foreign key constraint. Oracle Database XE does not do this for you automatically.

See "Ensuring Data Integrity in Tables With Constraints" on page 2-13.

Guidelines for Creating Indexes

You can create indexes on columns to speed up queries. Indexes provide faster access to data for operations that return a small portion of the rows of in a table.

You can create an index on any column; however, if the column is not used in any of these situations, creating an index on the column does not increase performance, and the index takes up resources unnecessarily.

This section contains the following topics:

- Index the Correct Tables and Columns on page 2-28
- Limit the Number of Indexes for Each Table on page 2-29
- Drop Indexes That Are No Longer Required on page 2-29

Index the Correct Tables and Columns

Use the following guidelines to determine when to create an index on a table or column:

- Create an index on the columns that are used for joins to improve join performance.
- You might want to create an index on a foreign key. See "Foreign Key Constraint" on page 2-16 for more information.
- Small tables do not require indexes. However, if a query is taking too long, then the table might have grown.

Columns with one or more of the following characteristics are good candidates for indexing:

- Values in the column are unique, or there are few duplicate values.
- There is a wide range of values.
- The column contains many nulls, but queries often select all rows that have a value.

Columns that contain many null values are less suitable for indexing if you do not search on the non-null values.

Limit the Number of Indexes for Each Table

The more indexes, the more overhead is incurred as the table is altered. When rows are inserted or deleted, all indexes on the table must be updated. When a column is updated, all indexes on the column must be updated.

You must weigh the performance benefit of indexes for queries against the performance overhead of updates. For example, if a table is primarily read-only, you might use more indexes; but, if a table is heavily updated, you might use fewer indexes.

Drop Indexes That Are No Longer Required

You might drop an index if:

- It does not speed up queries. The table might be very small, or there might be many rows in the table but very few index entries.
- The queries in your applications do not use the index.

You cannot drop an index that was created through a constraint. You must drop the constraint and then the index is dropped also.

If you drop a table, then all associated indexes are dropped. To drop an index, the index must be contained in your schema or you must have the `DROP ANY INDEX` system privilege.

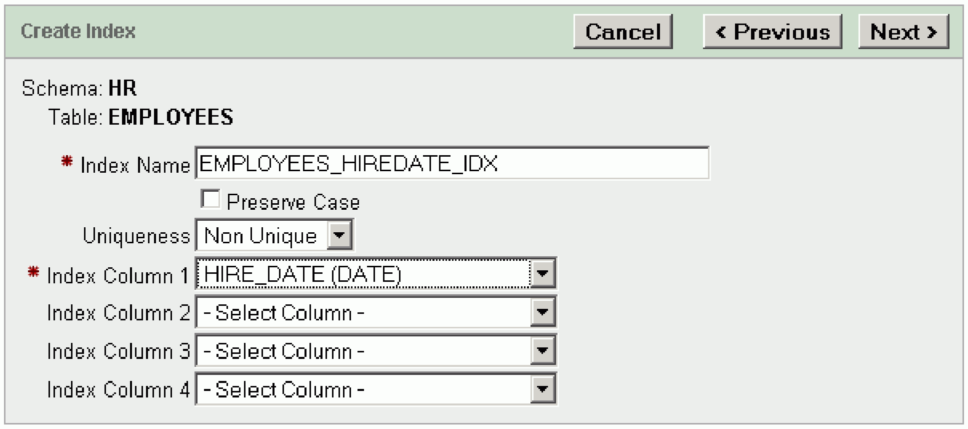
Creating an Index

You can create an index with the Object Browser page. To create an index, you specify one or more columns to be indexed and the type of index you want to create.

In the following example, an index is created on the `hire_date` column of the `employees` table. When the `hire_date` column is used as a condition for retrieving data, an index on that column increases the speed of those queries.

To create an index:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Detail pane, select **Index** from the **Create** menu.
4. In the **Table Name** field, enter `employees`.
5. Set the **Type of Index** to `Normal`, then click the **Next** button. See "Index Types" on page 2-27.
6. In the **Index Name** field, enter `EMPLOYEES_HIREDATE_IDX`.
7. Do not check the **Preserve Case** box.
8. Ensure that **Uniqueness** is set to `Non Unique`. The `hire_date` column can have duplicate values.
9. In the **Index Column 1** list, select `HIRE_DATE`, then click the **Next** button.



Schema: **HR**
Table: **EMPLOYEES**

* Index Name: EMPLOYEES_HIREDATE_IDX
☐ Preserve Case
Uniqueness: Non Unique
* Index Column 1: HIRE_DATE (DATE)
Index Column 2: - Select Column -
Index Column 3: - Select Column -
Index Column 4: - Select Column -

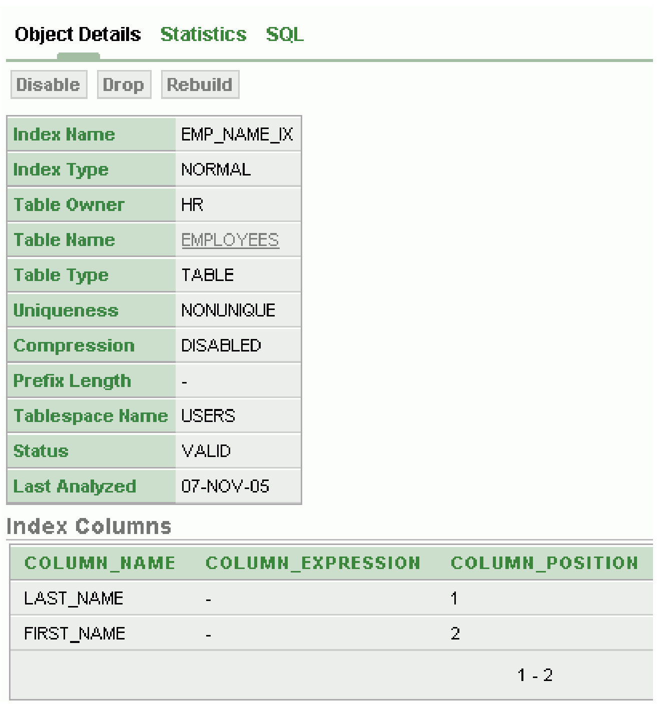
10. Click the **SQL** button to view the SQL statement that creates the index.
11. Click the **Finish** button to complete the action.

Displaying an Index for a Table

You can use Object Browser to display information about an index on a specific table.

To display information for an index:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Tables**, then click the `employees` tables.
4. Click **Indexes** to display the indexes for the table.
5. In the **Index** list, click the `EMP_NAME_IX` index to display details for that index. After viewing the object details for the index, click the **SQL** tab to display the SQL statement used to create the index.



Index Name	EMP_NAME_IX
Index Type	NORMAL
Table Owner	HR
Table Name	EMPLOYEES
Table Type	TABLE
Uniqueness	NONUNIQUE
Compression	DISABLED
Prefix Length	-
Tablespace Name	USERS
Status	VALID
Last Analyzed	07-NOV-05

Index Columns

COLUMN_NAME	COLUMN_EXPRESSION	COLUMN_POSITION
LAST_NAME	-	1
FIRST_NAME	-	2

1 - 2

6. Click other indexes in the Object list to display information about those indexes.

Dropping an Index

If you no longer need an index, you can use the Object Browser page to drop the index. See "Drop Indexes That Are No Longer Required" on page 2-29.

To test this procedure, follow the procedure in "Creating an Index" on page 2-29 to create an index.

To drop an index:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Indexes**, then click the `EMPLOYEES_HIREDATE_IDX` that you previously created.
4. Click the **Drop** button to drop the selected index.
5. Click the **Finish** button to complete the action.

Note: You cannot drop an index that is currently used to enforce a constraint. You must disable or drop the constraint and then, if the index is not dropped as a result of that action, drop the index.

Managing Views

Views are customized presentations of data in one or more tables or other views. You can think of them as stored queries. Views do not actually contain data, but instead derive their data from the tables upon which they are based. These tables are referred to as the base tables of the view.

As with tables, views can be queried, updated, inserted into, and deleted from, with some restrictions. All operations performed on a view affect the base tables of the view. Views provide an additional level of security by restricting access to a predetermined set of rows and columns of a table. They also hide data complexity and store complex queries.

This section contains the following topics:

- Creating a View on page 2-32
- Displaying a View on page 2-33
- Dropping a View on page 2-33

See Also: *Oracle Database Express Edition Application Express User's Guide* for information about managing views

Creating a View

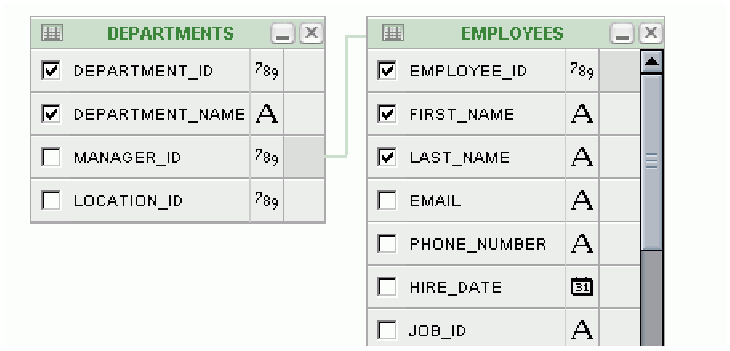
You can use Object Browser to create a view. The following example creates a view derived from the `departments` and `employees` tables to display department information along with the corresponding name of the manager.

This view combines the `department_id`, `department_name`, and `manager_id` columns from the `departments` table with the `employee_id`, `first_name`, and `last_name` columns of the `employees` table.

The tables are joined from the `manager_id` of the `departments` table to the `employee_id` of the `employees` table. This ensures that the corresponding first and last name of a manager is displayed in the view.

To create a view:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Detail pane, select **Views** from the **Create** menu.
4. In the **View Name** field, enter the name of the view (`my_emp_view`).
5. Click **Query Builder** to build the query for the view.
6. Click the `departments` table, and select the `department_id` and `department_name` columns.
7. Click the `employees` table, and select the `employee_id`, `first_name`, and `last_name` columns.
8. Click the blank box to the right of `manager_id` in the `departments` table to choose this column for a join with the `employees` table.
9. Click the blank box to the right of the `employee_id` in the `employees` table to choose this as the corresponding column for the join with `manager_id` of the `departments` table. Note the line that is added to the diagram connecting the two tables.



10. Click the **Run** button to see the results of querying this view.
11. Click the **Return** button to return to Object Browser.
12. Click the **Next** button.
13. Click the **SQL** button to view the SQL statement that creates the view.
14. Click the **Create** button to create the view.

Displaying a View

You can use Object Browser to display information about a view.

To display information about a view:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Views** then click the `EMP_DETAILS_VIEW` view.
4. Click the **SQL** button to view the SQL statement that created the view.
5. Click the **Data** button to view the data displayed in the view.

Dropping a View

If you no longer need a view, then you can use the Object Browser page to drop the view. To test this procedure, follow the procedure in "Creating a View" on page 2-32 to create a view.

To drop a view:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Views** then click the `my_emp_view` view that you previously created.
4. Click the **Drop** button.
5. Click the **Finish** button to complete the action.

Managing Sequences

A sequence is a database object that generates unique sequential values. These values are often used for primary and unique keys. Using a sequence generator to provide the value for a primary key in a table guarantees that the key value is unique.

You can refer to sequence values in SQL statements with these pseudocolumns:

- `CURRVAL`: Returns the current value of a sequence
- `NEXTVAL`: Increments the sequence and returns the next value

You must qualify `CURRVAL` and `NEXTVAL` with the name of the sequence, such as `employees_seq.CURRVAL` or `employees_seq.NEXTVAL`.

When you create a sequence, you can define its initial value and the increment between its values. The first reference to `NEXTVAL` returns the initial value of the sequence. Subsequent references to `NEXTVAL` increment the sequence value by the defined increment and return the new value. Any reference to `CURRVAL` returns the current value of the sequence, which is the value returned by the last reference to `NEXTVAL`.

Before you use `CURRVAL` for a sequence in your session, you must first initialize the sequence with `NEXTVAL`.

This section contains the following topics:

- Creating a Sequence on page 2-34
- Displaying a Sequence on page 2-35
- Dropping a Sequence on page 2-35

For examples of managing sequences using SQL statements, see "Creating and Dropping a Sequence With SQL" on page 3-26.

See Also: *Oracle Database Express Edition Application Express User's Guide* for information about managing sequences

Creating a Sequence

You can use the Object Browser page to create a sequence.

To create a sequence:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Detail pane, select **Sequence** from the **Create** menu.
4. In the **Sequence Name** field, enter the name of the new sequence (`my_sequence`).
5. In the **Start With** field, enter `1000`. This starts the sequence with a value of 1000.
6. For the other fields on the page, use the default values. Click the **Next** button.
7. Click the **SQL** button to view the SQL statement that creates this sequence.
8. Click the **Finish** button to create the sequence.

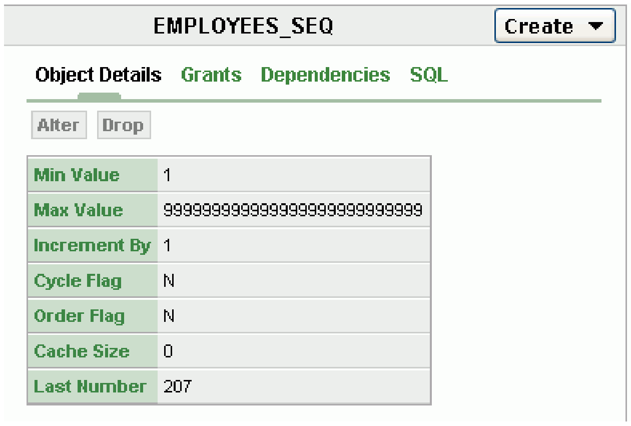
After creating and initializing a sequence, you can access and use the current value of the sequence. For an example of the use of a sequence in a SQL statement to insert data into a table, see Example 3–42 on page 3-26.

Displaying a Sequence

You can use the Object Browser page to display information about a sequence.

To display information about a sequence:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Sequences** then click the `EMPLOYEES_SEQ` sequence that was created for use with the `employees` table.



Min Value	1
Max Value	9999999999999999999999999999
Increment By	1
Cycle Flag	N
Order Flag	N
Cache Size	0
Last Number	207

4. Click other sequences in the Object list to display information about those sequences.

Dropping a Sequence

You can use the Object Browser page to drop a sequence. To test this procedure, follow the procedure in "Creating a Sequence" on page 2-34 to create a sequence.

To drop a sequence:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Sequences** then click the `my_sequence` that you previously created.
4. Click the **Drop** button to drop the selected sequence.
5. Click the **Finish** button to complete the action.

Managing Synonyms

A synonym is an alias for any schema object such as a table or view. Synonyms provide an alternative name for a database object and can be used to simplify SQL statements for database users. For example, you can create a synonym named `emps` as an alias for the `employees` table in the `HR` schema.

If a table in an application has changed, such as the `personnel` table has replaced the `employees` table, you can use the `employees` synonym to refer to the `personnel` table so that the change is transparent to the application code and the database users.

Because a synonym is simply an alias, it does not require any storage in the database other than its definition.

You can create both public and private synonyms. A public synonym can be accessed by every user in a database. A private synonym is in the schema of a specific user who has control over its availability to others.

This section contains the following topics:

- Creating a Synonym on page 2-36
- Displaying a Synonym on page 2-36
- Dropping a Synonym on page 2-36

For examples of managing synonyms using SQL statements, see "Creating and Dropping a Synonym With SQL" on page 3-27.

> **See Also:** *Oracle Database Express Edition Application Express User's Guide* for information about managing synonyms

Creating a Synonym

You can use the Object Browser page to create a synonym.

To create a synonym:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Detail pane, select **Synonym** from the **Create** menu.
4. In the **Synonym Name** field, enter the name of the synonym (`emps`).
5. In the **Object** field, enter `employees`.
6. For the other fields on the page, use the default values. Click the **Next** button.
7. Click the **SQL** button to see the SQL statement that creates this sequence.
8. Click the **Finish** button to create the synonym.

Displaying a Synonym

You can use the Object Browser page to display information about a synonym.

To display information about a synonym:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, select **Synonyms** then click the `emps` synonym that you previously created to display object details for that synonym.

Dropping a Synonym

You can use the Object Browser page to drop a synonym. To test this procedure for dropping a synonym, follow the procedure in "Creating a Synonym" on page 2-36 to create a synonym.

To drop a synonym:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Object list, Select **Synonyms** then click the `emps` synonym that you previously created.
4. Click the **Drop** button to drop the selected synonym.
5. Click the **Finish** button to complete the action.

3 Using SQL

This section discusses how to use Structured Query Language (SQL) with Oracle Database Express Edition, including how to retrieve and manipulate data, use SQL functions, and create database objects.

This section contains the following topics:

See Also:

- *Oracle Database SQL Reference* for detailed information about SQL statements and other parts of SQL, such as operators, functions, and format models
- *Oracle Database Concepts* for conceptual information about SQL
- *SQL*Plus User's Guide and Reference* for information about SQL*Plus, Oracle's version of SQL
- *Oracle Database Sample Schemas* for information about the `HR` sample schema that is used for examples in this chapter

Overview of SQL

SQL is nonprocedural language for accessing a database. You run SQL statements commands to perform various tasks, such as retrieving data from tables in Oracle Database XE. The SQL language automatically handles how to navigate the database and perform the desired task. All database operations are performed using SQL statements.

With SQL statements you can perform the following:

- Query, insert, and update data in tables
- Format, perform calculations on, store, and print from query results
- Examine table and object definitions

Oracle SQL statements are divided into several categories:

- Data Manipulation Language (DML) statements

  These statements query, insert, update, and delete data in tables.

- Transaction Control statements

  These statements commit or roll back the processing of transactions. A group of changes that you make is referred to as a transaction.

- Data Definition Language (DDL) statements

  These statements create, alter, and drop database objects.

A statement consists partially of SQL reserved words, which have special meaning in SQL and cannot be used for any other purpose. For example, `SELECT` and `UPDATE` are reserved words and cannot be used as table names. For a list of SQL reserved, see Appendix B, "Reserved Words".

A SQL statement is an instruction. The statement must be the equivalent of a complete SQL sentence, for example:

```
SELECT last_name, department_id FROM employees;
```

See Also: *Oracle Database SQL Reference* for more information about the types of SQL statements

Running SQL Statements

You can enter and run SQL statements with the SQL Commands page, Script Editor page, or SQL Command Line (SQL*Plus).

Using the SQL Commands and Script Editor pages are described in this section. The SQL Commands page is a simpler interface and easier to use.

Both SQL Commands and Script Editor pages enable you to save your SQL statements as a script file in a database repository for future use. You can run multiple SQL statements in the Script Editor page. Script Editor also enables you to download the script to the local file system, which can be run as a SQL script with SQL Command Line. For information about running SQL statements or SQL scripts with SQL Command Line, see Appendix A, "Using SQL Command Line".

This section contains the following topics:

- Running SQL Statements on the SQL Commands Page on page 3-2
- Running SQL Statements in the Script Editor Page on page 3-3

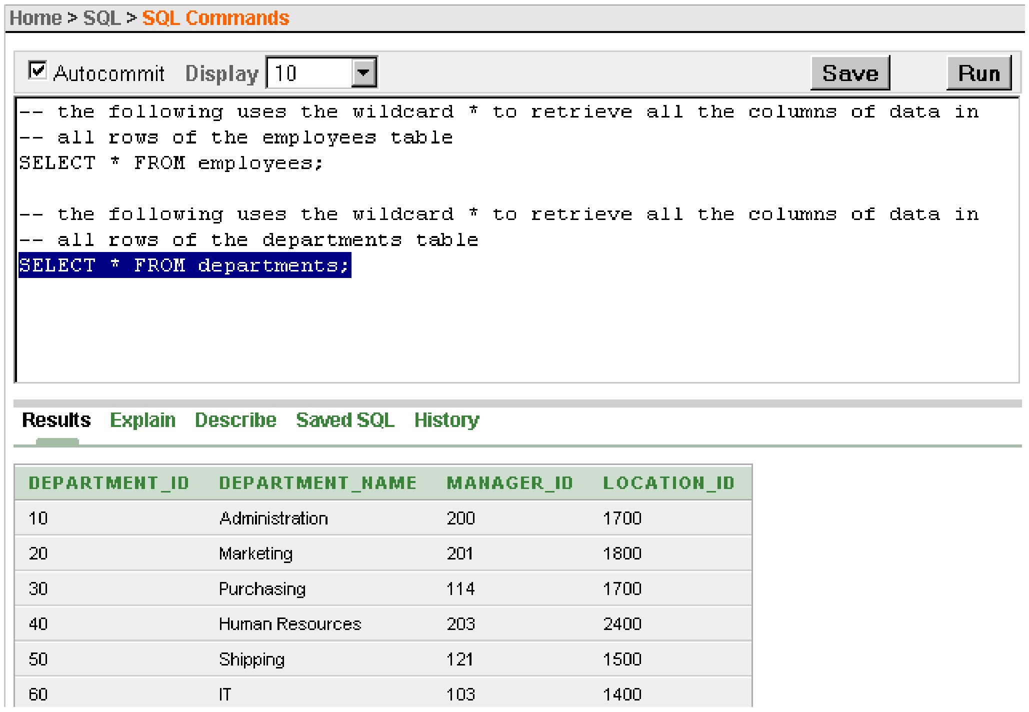
Running SQL Statements on the SQL Commands Page

To enter and run SQL statements in the SQL Commands page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user `HR` with your password for the `HR` account.
2. On the Database Home Page, click the **SQL** icon to display the SQL page.
3. Click the **SQL Commands** icon to display the SQL Commands page.
4. On the SQL Commands page, enter the SQL statements in Example 3–1 on page 3-5. Note that SQL statements are terminated with a semi colon (;) in the examples. The semi colon is required when running the SQL statements in a SQL

script or at the SQL Command Line prompt, but it is optional on the SQL Commands page.

5. Select (highlight) the SQL statement that you want to run, then click **Run** to run the statement and display the results.



```
-- the following uses the wildcard * to retrieve all the columns of data in
-- all rows of the employees table
SELECT * FROM employees;

-- the following uses the wildcard * to retrieve all the columns of data in
-- all rows of the departments table
SELECT * FROM departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400

6. If you want to save the SQL statements for future use, click the **Save** button.

7. In the **Name** field, enter a name for the saved SQL statements. You can also enter an optional description. Click the **Save** button to save the SQL statement.

8. To access saved SQL statements, click the **Saved SQL** tab and select the name of the saved SQL statement that you want to access.

> **See Also:** *Oracle Database Express Edition Application Express User's Guide* for detailed information about using SQL Commands

Running SQL Statements in the Script Editor Page

You can enter SQL statements on the Script Editor page and create a SQL script that can be saved in the database. The script can be downloaded to the local file system, and can be run from SQL Command Line (SQL*Plus). For information about running SQL scripts from SQL Command Line, see "Running Scripts From SQL Command Line" on page A-4.

To access and run SQL statements on the SQL Script Editor page:

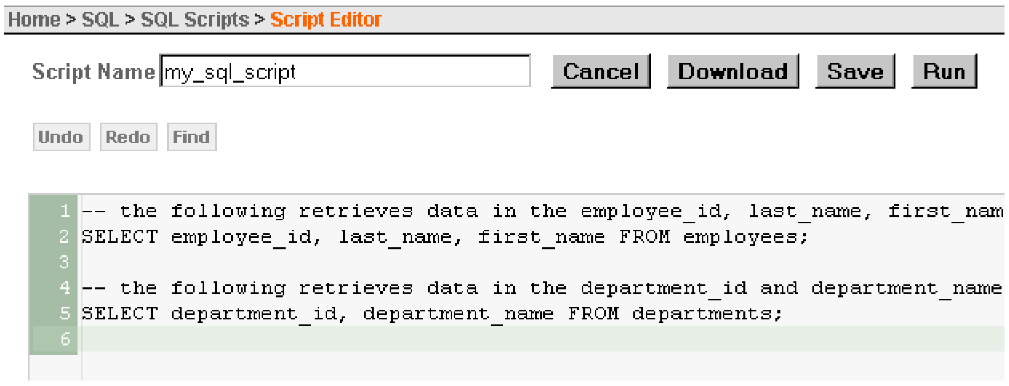
1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user `HR` with your password for the `HR` account.

2. On the home page, click the **SQL** icon to display the SQL page.

3. Click the **SQL Scripts** icon to display the Script Editor page.

4. Click the **Create** button to create a SQL script.

5. In the **Script Name** field, enter a name (`my_sql_script`) for the script.
6. In the **Script Editor** entry area, enter the SQL statements and comments in Example 3–2 on page 3-6.

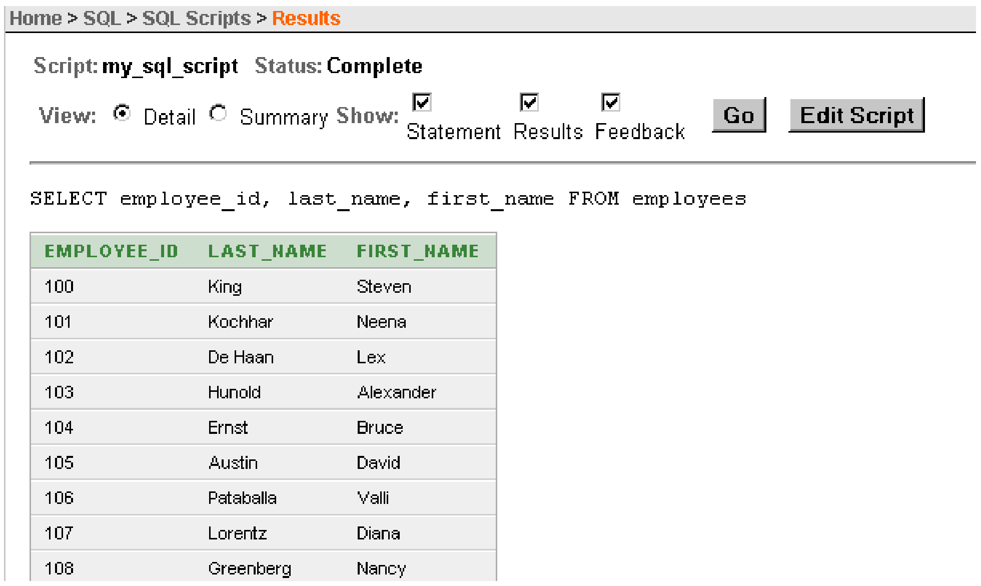


Script Name: my_sql_script

```
-- the following retrieves data in the employee_id, last_name, first_nam
SELECT employee_id, last_name, first_name FROM employees;

-- the following retrieves data in the department_id and department_name
SELECT department_id, department_name FROM departments;
```

7. Click the **Run** button on the Script Editor page to begin the processing of the statements in the script.

   The Run Script page displays information about the script, including any errors or SQL Command Line (SQL*Plus) commands that will be ignored when the script is run.
8. Click the **Run** button on the Run Script page to confirm your request, and start running the script.
9. Click the **View Results** icon for the script (`my_sql_script`) on the Manage Scripts page to display the results of the script.
10. Select the **Detail** view and enable all the **Show** options on the Results page to display details about the script results.



Script: my_sql_script Status: Complete

View: [✓ Detail] Summary Show: [✓ Statement] [✓ Results] [✓ Feedback]

```
SELECT employee_id, last_name, first_name FROM employees
```

EMPLOYEE_ID	LAST_NAME	FIRST_NAME
100	King	Steven
101	Kochhar	Neena
102	De Haan	Lex
103	Hunold	Alexander
104	Ernst	Bruce
105	Austin	David
106	Pataballa	Valli
107	Lorentz	Diana
108	Greenberg	Nancy

11. Click the **Edit Script** button to continue working on the SQL script.
12. When you are finished updating the script, click the **Save** button to save the script file in the database repository for future use.

13. To save the SQL script on the local file system, click the **Download** button, and choose the location for the script file. Note that the `.sql` extension is appended to the SQL script name.

> **See Also:** *Oracle Database Express Edition Application Express User's Guide* for detailed information about using SQL Scripts

Retrieving Data With Queries

You can retrieve data from rows stored in one or more database tables or views with a query using the SQL `SELECT` statement. The `SELECT` statement retrieves all of or part of the column data from rows depending on the conditions that you specify in the `WHERE` clauses. The group of columns that are selected from a table is referred to as the `SELECT` list.

This section contains the following topics:

- Displaying Data Using the SELECT Statement on page 3-5
- Using a Column Alias to Change Headings When Selecting Data on page 3-6
- Restricting Data Using the WHERE Clause on page 3-6
- Using Regular Expressions When Selecting Data on page 3-8
- Sorting Data Using the ORDER BY Clause on page 3-9
- Displaying Data From Multiple Tables on page 3-10
- Using Bind Variables With the SQL Commands Page on page 3-12

> **See Also:** *Oracle Database SQL Reference* for detailed information about the SQL `SELECT` statement

Displaying Data Using the SELECT Statement

With the SQL `SELECT` statement, you can query and display data in tables or views in a database.

Example 3–1 shows how to use `SELECT` to retrieve data from the `employees` and `departments` tables. In this example, the data for all columns in a row (record) of the tables are retrieved using the wildcard (*) notation. Note the use of comments to document the SQL statements. The comments (or remarks) in this example begin with two hyphens (`--`), but you can also use `rem` or `REM`.

Example 3–1 Using the SQL SELECT Statement to Query All Data From a Table

```
-- the following uses the wildcard * to retrieve all the columns of data in
-- all rows of the employees table
SELECT * FROM employees;

-- the following uses the wildcard * to retrieve all the columns of data in
-- all rows of the departments table
SELECT * FROM departments;
```

Example 3–2 shows how to use `SELECT` to retrieve data for specific columns of the `employees` and `departments` tables. In this example, you explicitly enter the column names in the `SELECT` statement. For information about the columns in the `employees` and `departments` table, see "Managing Database Objects With Object Browser" on page 2-2.

Example 3–2 Using the SQL SELECT Statement to Query Data From Specific Columns

```
-- the following retrieves data in the employee_id, last_name, first_name columns
SELECT employee_id, last_name, first_name FROM employees;

-- the following retrieves data in the department_id and department_name columns
SELECT department_id, department_name FROM departments;
```

Example 3–3 shows how to use `SELECT` to retrieve data from the `emp_details_view` view.

Example 3–3 Using the SQL SELECT Statement to Query Data in a View

```
-- the following retrieves all columns of data in all rows of the emp_details_view
SELECT * FROM emp_details_view;

-- the following retrieves data from specified columns in the view
SELECT employee_id, last_name, job_title, department_name, country_name,
       region_name FROM emp_details_view;
```

Using a Column Alias to Change Headings When Selecting Data

When displaying the result of a query, SQL normally uses the name of the selected column as the column heading. You can change a column heading by using a column alias to make the heading more descriptive and easier to understand.

You can specify the alias after the column name in the `SELECT` list using a space as a separator. If the alias contains spaces or special characters, such as number sign # or dollar sign $, or if it is case-sensitive, enclose the alias in quotation marks " ".

Example 3–4 shows the use of a column alias to provide a descriptive heading for each of the columns selected in a query.

Example 3–4 Using a Column Alias for a Descriptive Heading in a SQL Query

```
-- the following retrieves the data in employee_id, last_name, first_name columns
-- and provides column aliases for more descriptive headings of the columns
SELECT employee_id "Employee ID number", last_name "Employee last name",
  first_name "Employee first name" FROM employees;
```

Restricting Data Using the WHERE Clause

The `WHERE` clause uses comparison operators to identify specific rows in a table. When used with the `SELECT` statement, you can selectively retrieve rows from a table, rather than retrieving all rows of a table.

Comparison operators include those listed in Table 3–1.

Table 3–1 Comparison Operators

Operator	Definition
=, !=, <>	Test for equal to, not equal to, not equal to
>, >=, <, <=	Test for greater than, greater than or equal to, less than, less than or equal to
BETWEEN ... AND ...	Checks for a range between and including two values
LIKE	Searches for a match in a string, using the wildcard symbols % (zero or multiple characters) or _ (one character)
IN ( ), NOT IN ( )	Tests for a match, or not match, in a specified list of values

Table 3–1 (Cont.) Comparison Operators

Operator	Definition
IS NULL, IS NOT NULL	Checks whether a value is null, is not null

Example 3–5 shows how to use `SELECT` with a `WHERE` clause and several comparison operators to retrieve specific rows of data from the `employees` table.

Example 3–5 Selecting Data With the SQL WHERE Clause to Restrict Data

```
-- the following retrieves data where the manager_id equals 122
SELECT * FROM employees WHERE manager_id = 122;

-- this retrieves data where the manager_id equals 122 and job_id is ST_CLERK
SELECT * FROM employees WHERE manager_id = 122 AND job_id = 'ST_CLERK';

-- this retrieves employees with managers with IDs between 122 and 125 inclusive
SELECT * FROM employees WHERE manager_id BETWEEN 122 AND 125;

-- this uses LIKE with the wildcard % to retrieve employee data
-- where the last name contains mar somewhere in the name string
SELECT employee_id, last_name FROM employees WHERE last_name LIKE '%mar%';

-- this uses LIKE with the wildcard % to retrieve employee data
-- from the employees table where the last name starts with Mar
SELECT employee_id, last_name FROM employees WHERE last_name LIKE 'Mar%';

-- this retrieves employee data where the commission percentage is not null
SELECT employee_id, last_name FROM employees WHERE commission_pct IS NOT NULL;

-- the following retrieves data where the employee_id equals 125, 130, or 135
SELECT employee_id, last_name, first_name FROM employees
       WHERE employee_id IN (125, 130, 135);
```

See Also: *Oracle Database SQL Reference* for detailed information about using the `WHERE` clause

Using Character Literals in SQL Statements

Many SQL statements contain conditions, expressions, and functions that require you to specify character literal values. By default, you must use single quotation marks with character literals, such as `'ST_CLERK'` or `'Mar%'`. This technique can sometimes be inconvenient if the text itself contains single quotation marks. In such cases, you can also use the quote-delimiter mechanism, which enables you to specify `q` or `Q` followed by a single quotation mark and then another character to be used as the quotation mark delimiter.

The quote-delimiter can be any single-byte or multi-byte character except for a space, tab, or return. If the opening quote-delimiter is a left bracket `[`, left brace `{`, left angle bracket <, or left parenthesis `(` character, then the closing quote delimiter must be the corresponding right bracket `]`, right brace `}`, right angle bracket >, or right parenthesis `)` character. In all other cases, the opening and closing delimiter must be identical.

The following character literals use the alternative quoting mechanism:

```
q'(name LIKE '%DBMS_%%')'
q'#it's the "final" deadline#'
q'<'Data,' he said, 'Make it so.'>'
```

```
q'"name like '['"'
```

You can specify national character literals for unicode strings with the `N'text'` or `n'text'` notation, where `N` or `n` specifies the literal using the national character set. For example, `N'résumé'` is a national character literal. For information about unicode literals, see "Unicode String Literals" on page 7-21.

See Also:

- *Oracle Database Globalization Support Guide* for information about national character sets
- *Oracle Database SQL Reference* for information about character literals

Using Regular Expressions When Selecting Data

Regular expressions enable you to search for patterns in string data by using standardized syntax conventions. A regular expression can specify complex patterns of character sequences.

You specify a regular expression with metacharacters and literals. Metacharacters are operators that specify search algorithms. Literals are the characters for which you are searching.

The regular expression functions and conditions include `REGEXP_INSTR`, `REGEXP_LIKE`, `REGEXP_REPLACE`, and `REGEXP_SUBSTR`. Example 3–6 shows some examples of the use of the regular expression functions and conditions.

Example 3–6 Using Regular Expressions With the SQL SELECT Statement

```
-- in the following example, the REGEXP_LIKE is used to select rows where
-- the value of job_id starts with ac, fi, mk, or st,
-- then follows with _m, and ends with an or gr
-- the metacharacter | specifies OR
-- the 'i' option specifies case-insensitive matching
SELECT employee_id, job_id FROM employees
   WHERE REGEXP_LIKE (job_id, '[ac|fi|mk|st]_m[an|gr]', 'i');

-- in the following example, REGEXP_REPLACE is used to replace
-- phone numbers of the format "nnn.nnn.nnnn" with
-- parentheses, spaces, and dashes to produce this format "(nnn) nnn-nnnn"
-- digits (0-9) are denoted with the metacharacter [:digit:]
-- the metacharacter {n} specifies a fixed number of occurrences
-- the \ is used an escape character so that the subsequent metacharacter
-- in the expression is treated as a literal, such as \.; otherwise, the
-- metacharacter . denotes any character in the expression
SELECT phone_number, REGEXP_REPLACE( phone_number,
 '([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{4})', '(\1) \2-\3')
 "Phone Number" FROM employees;

-- in the following example, REGEXP_REPLACE is used to replace
-- phone numbers of the format "nnn.nnn.nnnn.nnnnnn"
-- with the format "+nnn-nn-nnnn-nnnnnn"
SELECT phone_number, REGEXP_REPLACE( phone_number,
 '([[:digit:]]{3})\.([[:digit:]]{2})\.([[:digit:]]{4})\.([[:digit:]]{6})',
 '+\1-\2-\3-\4') "Phone Number" FROM employees;

-- in the following example, REGEXP_SUBSTR returns the first substring
-- composed of one or more occurrences of digits and dashes
-- the metacharacter + specifies multiple occurrences in [[:digit:]-]+
SELECT street_address, REGEXP_SUBSTR(street_address, '[[:digit:]-]+', 1, 1)
  "Street numbers" FROM locations;

-- in the following example, REGEXP_INSTR starts searching at the first character
-- in the string and returns the starting position (default) of the second
-- occurrence of one or more non-blank characters
-- REGEXP_INSTR returns 0 if not found
-- the metacharacter ^ denotes NOT, as in NOT space [^ ]
SELECT street_address, REGEXP_INSTR(street_address, '[^ ]+', 1, 1)
  "Position of 2nd block" FROM locations;
```

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals* for information about using regular expressions
- *Oracle Database SQL Reference* for information about regular expression metacharacters
- `REGEXP_INSTR`, `REGEXP_LIKE`, `REGEXP_REPLACE`, and `REGEXP_SUBSTR` in *Oracle Database SQL Reference*

Sorting Data Using the ORDER BY Clause

You can use `SELECT` with the `ORDER BY` clause to retrieve and display rows from a table ordered (sorted) by a specified column in the table. The specified column in the `ORDER BY` clause does not have to be in the `SELECT` list of columns that you want to display.

You can specify the sort order as `ASC` for ascending or `DESC` for descending. The default sort order is ascending, which means:

- Numeric values are displayed with the lowest values first, such as 1 to 999.
- Character values are displayed in alphabetical order, such as A first and Z last.
- Date values are displayed with the earliest value first, such as 01-JUN-93 before 01-JUN-95.

Null (empty) values are displayed last for ascending sequences and first for descending sequences.

Example 3–7 shows how to use `SELECT` with the `ORDER BY` clause to retrieve and display rows from the `employees` table ordered (sorted) by specified columns.

Example 3–7 Selecting Data With the SQL ORDER BY Clause to Sort the Data

```
-- the following retrieves rows with manager_id = 122 ordered by employee_id
-- the order is the default ascending order, lowest employee_id displays first
SELECT * FROM employees WHERE manager_id = 122 ORDER BY employee_id;

-- the following retrieves rows ordered by manager_id
-- the order is specified as descending, highest manager_id displays first
SELECT employee_id, last_name, first_name, manager_id FROM employees
      ORDER BY manager_id DESC;
```

See Example 3–23 on page 3-17 for the use of `ORDER BY` with the `GROUP BY` clause.

See Also: *Oracle Database SQL Reference* for detailed information about using `ORDER BY` with `SELECT`

Displaying Data From Multiple Tables

You can use SELECT to display data from multiple tables. This process is referred to as joining tables. In a join, rows from multiple tables are usually linked by similar columns.

Joining tables is useful when you need to view data that is stored in multiple tables. For example, the employees table contains employee information with a column of department IDs, but not the department names. The departments table contains columns for department IDs and names. By joining the tables on the department ID, you can view an employee's information with the corresponding department name.

There are several types of joins, including self, inner, and outer. A self-join joins a table to itself. Example 3–11 on page 3-11 is an example of a self- join. An inner join (sometimes called a simple join) is a join of two or more tables that returns only those rows that satisfy the join condition. Any unmatched rows are not displayed in the output. Example 3–8 on page 3-10 and Example 3–9 on page 3-10 are examples of inner joins. An outer join extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition. There are three types of outer joins: LEFT OUTER, RIGHT OUTER, and FULL OUTER. Example 3–12 on page 3-11 shows examples of a outer joins.

When you retrieve data from multiple tables, you can explicitly identify to which table a column belongs. This is important when tables contain columns with the same name. You can use the complete table name to explicitly identify a column, such as employees.employee_id, or a table alias. Note the use of the table aliases (d, e, and l) to explicitly identify the columns by table in the SQL statement in Example 3–9 and Example 3–10. The alias is defined in the FROM clause of the SQL statement. A table alias is used, rather than the table name, to simplify and reduce the size of the SQL code.

You can join two tables automatically on all the columns that have matching names and datatypes using the NATURAL JOIN syntax as shown in Example 3–8. This join select rows from the two tables that have equal values in the matched columns. If the columns with the same name have different datatypes, an error results.

Example 3–8 Selecting Data From Two Tables With the SQL NATURAL JOIN Syntax

```
-- the following SELECT statement retrieves data from two tables
-- that have a corresponding column(s) with equal values
-- for employees and departments, matching columns are department_id, manager_id
SELECT employee_id, last_name, first_name, department_id,
  department_name, manager_id  FROM employees
  NATURAL JOIN departments;
```

Example 3–9 is an example of querying data from joined tables using the JOIN ... USING syntax. The first SELECT joins two tables, and the second SELECT joins three tables. With the JOIN ... USING syntax, you explicitly specify the join columns. The columns in the tables that are used for the join must have the same name. Note that the table alias is not used on the referenced columns.

Example 3–9 Selecting Data From Multiple Tables WIth the SQL JOIN USING Syntax

```
-- the following SELECT statement retrieves data from two tables
-- that have a corresponding column (department_id)
-- note that the employees table has been aliased to e and departments to d
SELECT e.employee_id, e.last_name, e.first_name, e.manager_id, department_id,
  d.department_name, d.manager_id FROM employees e
  JOIN departments d USING (department_id);
```

```
-- the following SELECT retrieves data from three tables
-- two tables have the corresponding column (department_id) and
-- two tables have the corresponding column (location_id)
SELECT e.employee_id, e.last_name, e.first_name, e.manager_id, department_id,
  d.department_name, d.manager_id, location_id, l.country_id FROM employees e
  JOIN departments d USING (department_id)
  JOIN locations l USING (location_id);
```

Example 3–10 is an example of querying data from joined tables using `JOIN` ... `ON` syntax. The first `SELECT` joins two tables, and the second `SELECT` joins three tables. Using the `ON` clause enables you to specify a join condition outside a `WHERE` clause and a join condition with columns that have different name, but equal values.

Example 3–10 Selecting Data From Multiple Tables With the SQL JOIN ON Syntax

```
-- the following SELECT statement retrieves data from two tables
-- that have a corresponding column department_id
-- note that the employees table has been aliased to e and departments to d
SELECT e.employee_id, e.last_name, e.first_name, e.department_id,
  d.department_name, d.manager_id FROM employees e
  JOIN departments d ON e.department_id = d.department_id
  WHERE e.manager_id = 122;

-- the following SELECT retrieves data from three tables
-- two tables have the corresponding column department_id and
-- two tables have the corresponding column location_id
SELECT e.employee_id, e.last_name, e.first_name, e.department_id,
  d.department_name, d.manager_id, d.location_id, l.country_id FROM employees e
  JOIN departments d ON e.department_id = d.department_id
  JOIN locations l ON d.location_id = l.location_id
  WHERE l.location_id = 1700;
```

You can join a table to itself, a process called a self-join. For example, if you want to view an employee ID and employee last name with the manager ID and manager name of that employee, you would use a self-join on the employees table as shown in Example 3–11. The `employees` table is joined to itself using the manager ID of the employee and employee ID of the manager. Note that the columns used for the join have different names. Column aliases, such as `emp_id` and `emp_lastname`, were used to clearly identify the column values in the output.

Example 3–11 Self Joining a Table With the SQL JOIN ON Syntax

```
-- the following SELECT statement retrieves data from the employees table
-- to display employee_id and last_name, along with manager_id and last_name
-- of the employee in a self-join
-- note that the employees table has been aliased to e and m
SELECT e.employee_id emp_id, e.last_name emp_lastname, m.employee_id mgr_id,
  m.last_name mgr_lastname
  FROM employees e
  JOIN employees m ON e.manager_id = m.employee_id;
```

Example 3–12 shows how to use outer joins.

Example 3–12 Using SQL Outer Joins

```
-- the following uses a LEFT OUTER JOIN
-- all rows are retrieved from the left table (employees) even if
-- there is no match in the right table (departments)
SELECT e.employee_id, e.last_name, e.department_id, d.department_name
```

```
  FROM employees e LEFT OUTER JOIN departments d
  ON (e.department_id = d.department_id);

-- the following uses a RIGHT OUTER JOIN
-- all rows are retrieved from the right table (departments) even if
-- there is no match in the left table (employees)
SELECT e.employee_id, e.last_name, d.department_id, d.department_name
  FROM employees e RIGHT OUTER JOIN departments d
  ON (e.department_id = d.department_id);

-- the following uses a FULL OUTER JOIN
-- all rows are retrieved from the employees table even if there is no match in
-- the departments table, and all rows are retrieved from the departments table
-- even if there is no match in the left table
SELECT e.employee_id, e.last_name, d.department_id, d.department_name
  FROM employees e FULL OUTER JOIN departments d
  ON (e.department_id = d.department_id);
```

See Also: *Oracle Database SQL Reference* for information about using SELECT with multiple tables

Using Bind Variables With the SQL Commands Page

You can use bind variables with the SQL Commands page to prompt for values when running a SQL statement, rather than supplying the value when the statement is created. Bind variables are prefixed with a colon. You can choose any name for the bind variable name, such as `:b`, `:bind_variable`, or `:employee_id`. For example, you could enter and run the following statement in the SQL Commands page:

```
SELECT * FROM employees WHERE employee_id = :employee_id
```

When you run a statement with a bind variable in the SQL Commands page, a window opens prompting you for a value for the bind variable. After entering a value, click the **Submit** button. Note that you might need to configure your Web browser to allow the popup window to display.

For information about using bind variables with PL/SQL, see "Using Bind Variables With PL/SQL" on page 4-27.

See Also: "Using Bind Variables" in *Oracle Database Express Edition Application Express User's Guide*

Using Pseudocolumns, Sequences, and SQL Functions

With SQL built-in functions, you can manipulate character, numeric, and date data in SQL statements to change how the data is displayed or to convert the data for insertion in a column of a table. You can also perform operations on a collection of data with aggregate functions.

Pseudocolumns are built-in values that provide specific information with a query and are similar to functions without arguments. However, functions without arguments typically return the same value for every row in the result set, whereas pseudocolumns typically return a different value for each row.

This section contains the following topics:

- Using ROWNUM, SYSDATE, and USER Pseudocolumns With SQL on page 3-13
- Using Arithmetic Operators on page 3-13

- Using Numeric Functions on page 3-14
- Using Character Functions on page 3-14
- Using Date Functions on page 3-15
- Using Conversion Functions on page 3-16
- Using Aggregate Functions on page 3-17
- Using NULL Value Functions on page 3-18
- Using Conditional Functions on page 3-19

See Also: *Oracle Database SQL Reference* for detailed information about SQL functions

Using ROWNUM, SYSDATE, and USER Pseudocolumns With SQL

A pseudocolumn is similar to a table column, but is not stored in a table. A pseudocolumn returns a value, so it is similar to a function without argument. Oracle Database XE provides several pseudocolumns, such as the `ROWNUM`, `SYSDATE`, and `USER`. The `ROWNUM` pseudocolumn returns a number indicating the order in which Oracle Database XE selects the row in a query. `SYSDATE` returns the current date and time set for the operating system on which the database resides. `USER` returns the name of the user name that is currently logged in.

Example 3–13 shows the use of the `SYSDATE` pseudocolumn. Note the use of the `DUAL` table, which is automatically created by Oracle Database XE for use as a dummy table in SQL statements. See Example 3–19 on page 3-15 for another example of the use of `SYSDATE`.

Example 3–13 Using the SQL SYSDATE Pseudocolumn

```
-- the following statement displays the SYSDATE, which is the current system date
-- NOW is a column alias for display purposes
-- DUAL is a dummy table with one row simply used to complete the SELECT statement
SELECT SYSDATE "NOW" FROM DUAL;
```

Example 3–14 shows the use of the `USER` pseudocolumn.

Example 3–14 Using the SQL USER Pseudocolumn

```
-- display the name of the current user, the user name should be HR
SELECT USER FROM DUAL;
```

Example 3–15 shows the use of the `ROWNUM` pseudocolumn.

Example 3–15 Using the SQL ROWNUM Pseudocolumn

```
-- using ROWNUM < 10 limits the number of rows returned to less than 10
SELECT employee_id, hire_date, SYSDATE FROM employees WHERE ROWNUM < 10;
```

Using Arithmetic Operators

You can use arithmetic operators to create expressions for calculations on data in tables. The arithmetic operators include:

- Plus sign + for addition
- Minus sign - for subtraction
- Asterisk * for multiplication

- Slash / for division

In an arithmetic expression, multiplication and division are evaluated first, then addition and subtraction. When operators have equal precedence, the expression is evaluated left to right. It is best to include parentheses to explicitly determine the order of operators and provide clarity in the expression.

Example 3–16 shows the use of arithmetic operators in expressions with the data in the `employees` table. Note the use of a column alias to provide a more descriptive heading for the displayed output.

Example 3–16 Using SQL Arithmetic Operators

```
-- in the following query the commission is displayed as a percentate instead
-- of the decimal that is stored in the database
SELECT employee_id, (commission_pct * 100) "Commission %" FROM employees;

-- in the following query, the proposed new annual salary is calculated
-- for employees who report to the manager with ID 145
SELECT employee_id, ((salary + 100) * 12) "Proposed new annual salary"
  FROM employees WHERE manager_id = 145;
```

Using Numeric Functions

Oracle Database XE provides a set of numeric functions that you can use in your SQL statements to manipulate numeric values. With numeric functions, you can round to a specified decimal, truncate to a specified decimal, and return the remainder of a division on numeric data.

Example 3–17 shows the use of numeric functions on numeric data.

Example 3–17 Using SQL Numeric Functions

```
-- you can use the ROUND function to round off numeric data, in this case to
-- two decimal places
SELECT employee_id, ROUND(salary/30, 2) "Salary per day" FROM employees;

-- you can use the TRUNC function to truncate numeric data, in this case to
-- 0 decimal places; 0 is the default so TRUNC(salary/30) would be same
SELECT employee_id, TRUNC(salary/30, 0) "Salary per day" FROM employees;

-- use the MOD function to return the remainder of a division
-- MOD is often used to determine is a number is odd or even
-- the following determines whether employee_id is odd (1) or even (0)
SELECT employee_id, MOD(employee_id, 2) FROM employees;
```

Using Character Functions

Oracle Database XE provides a set of character functions that you can use in your SQL statements to customize the character values. With character functions, you can perform operations that change the case, remove blanks, extract substrings from, replace substrings in, and concatenate character data.

Example 3–18 shows the use of some character functions on character data.

Example 3–18 Using SQL Character Functions

```
-- you can use the UPPER function to display uppercase data, LOWER for lowercase
SELECT employee_id, UPPER(last_name), LOWER(first_name) FROM employees;

-- you can use the INITCAP function to display uppercase only the first letter
```

```
SELECT employee_id, INITCAP(first_name), INITCAP(last_name) FROM employees;

-- you can use RTRIM and LTRIM to remove spaces from the beginning or end of
-- character data. Note the use of concatenation operator || to add a space
SELECT employee_id, RTRIM(first_name) || ' ' || LTRIM(last_name) FROM employees;

-- you can use TRIM to remove spaces from both the beginning and end
SELECT employee_id, TRIM(last_name) || ', ' || TRIM(first_name) FROM employees;

-- you can use RPAD to add spaces on the right to line up columns
-- in this case, spaces are added to pad the last_name output to 30 characters
SELECT employee_id, RPAD(last_name, 30, ' '), first_name FROM employees;

-- use SUBSTR to select a substring of the data, in the following only
-- the characters from 1 to 15 are selected from the last_name
SELECT employee_id, SUBSTR(last_name, 1, 10) FROM employees;

-- use LENGTH to return the number of characters in a string or expression
SELECT LENGTH(last_name) FROM employees;

-- use REPLACE to replace characters in a string or expression
SELECT employee_id, REPLACE(job_id, 'SH', 'SHIPPING') FROM employees
   WHERE SUBSTR(job_id, 1, 2) = 'SH';
```

Using Date Functions

Oracle Database Express Edition provides a set of date functions to manipulate and calculate date and time data. For example, with date functions you can add months to, extract a specific field from, truncate, and round a date value. You can also calculate the number of months between two dates.

Example 3–19 shows the use of some date functions on date data.

Example 3–19   Using SQL Date Functions

```
-- in the following statement you can use MONTHS_BETWEEN to compute months
-- employed for employees and then truncate the results to the whole month
-- note the use of the label (alias) "Months Employed" for the computed column
SELECT employee_id, TRUNC(MONTHS_BETWEEN(SYSDATE, HIRE_DATE)) "Months Employed"
  FROM employees;

-- the following extracts displays the year hired for each employee ID
SELECT employee_id, EXTRACT(YEAR FROM hire_date) "Year Hired" FROM employees;

-- the following extracts and concatenates the year, month, and day from SYSDATE
SELECT EXTRACT(YEAR FROM SYSDATE) || EXTRACT(MONTH FROM SYSDATE) ||
   EXTRACT(DAY FROM SYSDATE) "Current Date" FROM DUAL;

-- the following adds 3 months to the hire_date of an employee
SELECT employee_id, hire_date, ADD_MONTHS(hire_date, 3) FROM employees;

-- LAST_DAY finds the last day of the month for a specific date, such as hire_date
SELECT employee_id, hire_date, LAST_DAY(hire_date) "Last day of month"
   FROM employees;

-- the following returns the system date, including fractional seconds
-- and time zone, of the system on which the database resides
SELECT SYSTIMESTAMP FROM DUAL;
```

Using Conversion Functions

Oracle Database XE provides a set of conversion functions that for use in SQL statements to convert a value from one datatype to another datatype. For example, you can convert a character value to a numeric or date datatype or you can convert a numeric or date value to a character datatype. Conversion functions are useful when inserting values into a column of a table and when displaying data.

When converting a value, you can also specify a format model. A format model is a character literal that specifies the format of data. A format model does not change the internal representation of the value in the database.

Example 3–20 shows how to use the character conversion function with format models.

Example 3–20 Using the SQL Character Conversion Function

```
-- you can convert the system date (SYSDATE) to a character string and format
-- with various format models and then display the date as follows
SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY AD') "Today" FROM DUAL;
-- FM removes all leading or trailing blanks from Month
SELECT TO_CHAR(SYSDATE, 'FMMonth DD YYYY') "Today" FROM DUAL;
-- the following displays the system date and time with a format model
SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "Now" FROM DUAL;

-- you can convert and format a date column using format models
-- for Short or Long Date format
SELECT hire_date, TO_CHAR(hire_date,'DS') "Short Date" FROM employees;
SELECT hire_date, TO_CHAR(hire_date,'DL') "Long Date" FROM employees;

-- the following extracts the year, month, and date from SYSDATE, then converts
-- and formats the result with leading zeros and removes any leading blanks (FM)
SELECT TO_CHAR(EXTRACT(YEAR FROM SYSDATE)) ||
  TO_CHAR(EXTRACT(MONTH FROM SYSDATE),'FM09') ||
  TO_CHAR(EXTRACT(DAY FROM SYSDATE),'FM09') "Current Date" FROM DUAL;

-- the following returns the current date in the session time zone,
-- in a value in the Gregorian calendar of datatype DATE,
-- the returned value is converted to character and displayed with a format model
SELECT TO_CHAR(CURRENT_DATE, 'DD-MON-YYYY HH24:MI:SS') "Current Date" FROM DUAL;

-- you can convert and format numeric currency data as a character string
-- with a format model to add a $, commas, and deciaml point
SELECT TO_CHAR(salary,'$99,999.99') salary FROM employees;
```

Example 3–21 shows how to use the number conversion function.

Example 3–21 Using the SQL Number Conversion Function

```
-- you can convert a character string to a number
SELECT TO_NUMBER('1234.99') + 500 FROM DUAL;

-- the format model must match the format of the string you want to convert
SELECT TO_NUMBER('11,200.34', '99G999D99') + 1000 FROM DUAL;
```

Example 3–22 shows how to use some date conversion functions.

Example 3–22 Using SQL Date Conversion Functions

```
-- the following converts the character string to a date with
-- the specified format model
```

```
SELECT TO_DATE('27-OCT-98', 'DD-MON-RR') FROM DUAL;

-- the following converts the character string to a date with
-- the specified format model
SELECT TO_DATE('28-Nov-05 14:10:10', 'DD-Mon-YY HH24:MI:SS') FROM DUAL;

-- the following converts the character string to a date with
-- the specified format model
SELECT TO_DATE('January 15, 2006, 12:00 A.M.', 'Month dd, YYYY, HH:MI A.M.')
  FROM DUAL;

-- the following converts a character stirng to a timestamp with
-- the specified datetime format model
SELECT TO_TIMESTAMP('10-Sep-05 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF')
  FROM DUAL;
```

Be careful when using a date format such as `DD-MON-YY`. The `YY` indicates the year in the current century. For example, `31-DEC-92` is `December 31, 2092`, not `1992` as you might expect. If you want to indicate years in any century other than the current one, use a format model such as the default `RR`.

> **See Also:** *Oracle Database SQL Reference* for detailed information about format models

Using Aggregate Functions

Aggregate or group functions operate on sets of rows to give one result for each group. These sets can be the entire table or the table split into groups.

Example 3–23 shows how to use aggregate functions on collections of data in the database. Aggregate functions include `AVG`, `COUNT`, `DENSE_RANK`, `MAX`, `MIN`, `PERCENT_RANK`, `RANK`, `STDDEV`, and `SUM`. The `GROUP BY` clause is used to select groups of rows by a specified expression, and returns one row of summary information for each group. The `HAVING` clause is used to specify which groups to include, or exclude, from the output based on a group condition. The `DISTINCT` clause causes an aggregate function to consider only distinct values of the argument expression. The `ALL` clause, which is the default behavior, causes an aggregate function to consider duplicate values.

Example 3–23 Using SQL Aggregate Functions

```
-- you can use COUNT to count the employees with manager 122
-- note the use of a column alias Employee Count
SELECT COUNT(*) "Employee Count" FROM employees WHERE manager_id = 122;

-- count the employees grouped by manager, also sort the groups
SELECT COUNT(*) "Employee Count", manager_id  FROM employees
  GROUP BY manager_id
  ORDER BY manager_id;

-- count the number of employees that receive a commission
-- this returns the count where the commission_pct is not NULL
SELECT COUNT(commission_pct) FROM employees;

-- count the number of distinct department IDs assigned to the employees
-- this returns a number that does not include duplicates
SELECT COUNT(DISTINCT department_id) FROM employees;

-- you can use MIN, MAX, and AVG to find the minimum, maximum, and average
-- salaries for employees with manager 122
SELECT MIN(salary), MAX(salary), AVG(salary) FROM employees
  WHERE manager_id = 122;

-- this computes the minimum, maximum, and average salary by job ID groups
-- the job ID groups are sorted in alphabetical order
SELECT MIN(salary), MAX(salary), AVG(salary), job_id FROM employees
  GROUP BY job_id
  ORDER BY job_id;

-- the following returns the minimum and maximum salaries for employees grouped
-- by department for those groups having a minimum salary less than $7,000
SELECT department_id, MIN(salary), MAX (salary) FROM employees
   GROUP BY department_id
   HAVING MIN(salary) < 7000
   ORDER BY MIN(salary);

-- the following uses the PERCENT_RANK function to return the percent ranking
-- for a $11,000 salary among the employees who are managers
-- in this example, a percent ranking of 0 corresponds to the highest salary
SELECT PERCENT_RANK(11000) WITHIN GROUP
   (ORDER BY salary DESC) "Rank of $11,000 among managers"
   FROM employees WHERE job_id LIKE '%MAN' OR job_id LIKE '%MGR';

-- the following uses the RANK function to return the ranking for a $2,600 salary
-- among the employees who are clerks,
-- in this example a ranking of 1 corresponds to the highest salary in the group
SELECT RANK(2600) WITHIN GROUP
   (ORDER BY salary DESC) "Rank of $2,600 among clerks"
   FROM employees WHERE job_id LIKE '%CLERK';

-- the following uses RANK to show the ranking of SH_CLERK employees by salary
-- identical salary values receive the same rank and cause nonconsecutive ranks
SELECT job_id, employee_id, last_name, salary, RANK() OVER
   (PARTITION BY job_id ORDER BY salary DESC) "Salary Rank"
   FROM employees WHERE job_id = 'SH_CLERK';

-- the following uses DENSE_RANK to show the ranking of SH_CLERK employees
-- by salary, identical salary values receive the same rank and
-- rank numbers are consecutive (no gaps in the ranking)
SELECT job_id, employee_id, last_name, salary, DENSE_RANK() OVER
   (PARTITION BY job_id ORDER BY salary DESC) "Salary Rank (Dense)"
   FROM employees WHERE job_id = 'SH_CLERK';

-- the following computes the cumulative standard deviation of the salaries
-- for ST CLERKs ordered by hire_date
SELECT employee_id, salary, hire_date, STDDEV(salary)
   OVER (ORDER BY hire_date) "Std Deviation of Salary"
   FROM employees WHERE job_id = 'ST_CLERK';
```

Using NULL Value Functions

Oracle Database XE provides functions that you can use in your SQL statements to work with NULL values. For example, you can substitute a different value if value in a column of a table is NULL.

Example 3–24 shows the use of the SQL NVL function. This function substitutes the specified value when a NULL value is encountered.

Example 3–24 Using the SQL NVL Function

```
-- use the NVL function to substitute 0 for a NULL value in commission_pct
SELECT commission_pct, NVL(commission_pct, 0) FROM employees;

-- use the NVL function to substitute MISSING for a NULL value in phone_number
SELECT phone_number, NVL(phone_number, 'MISSING') FROM employees;
```

Example 3–25 shows the use of the SQL `NVL2` function. This function returns the second specified expression when the first expression is not `NULL`. If the first expression is `NULL`, the third expression is returned.

Example 3–25 Using the SQL NVL2 Function

```
-- use the NVL2 function to return salary + (salary * commission_pct)
-- if commission_pct is not NULL; otherwise, if commission_pct is NULL,
-- then return salary
SELECT employee_id , last_name, salary,
  NVL2(commission_pct, salary + (salary * commission_pct), salary) income
  FROM employees;
```

Using Conditional Functions

Oracle Database XE provides conditional functions that you can use in your SQL statements to return a value based on multiple search conditions values.

Example 3–26 shows the use of the SQL `CASE` functions.

Example 3–26 Using the SQL CASE Function

```
-- CASE can compare a column or expression or search conditions, returning
-- a result when there is a match. CASE is similar to IF_THEN-ELSE logic.
-- In the following, the value of the hire_date column is compared against various
-- dates. When there is a match, the corresponding calculated result is returned,
-- otherwise the default calculated salary is returned.
SELECT employee_id, hire_date , salary,
  CASE WHEN hire_date < TO_DATE('01-JAN-90') THEN salary*1.20
       WHEN hire_date < TO_DATE('01-JAN-92') THEN salary*1.15
       WHEN hire_date < TO_DATE('01-JAN-94') THEN salary*1.10
       ELSE salary*1.05 END  "Revised Salary"
  FROM employees;
```

Example 3–27 shows the use of the SQL `DECODE` functions.

Example 3–27 Using the SQL DECODE Function

```
-- DECODE compares a column or expression to search values, returning a result
-- when there is a match. DECODE is similar to IF_THEN-ELSE logic.
-- In the following, the value of the job_id column is compared against PU_CLERK,
-- SH_CLERK, and ST_CLERK.
-- When there is a match, the corresponding calculated result is returned,
-- otherwise the original salary is returned unchanged.
SELECT employee_id, job_id , salary,
  DECODE(job_id, 'PU_CLERK', salary*1.05,
                 'SH_CLERK', salary*1.10,
                 'ST_CLERK', salary*1.15,
                             salary) "Revised Salary"
  FROM employees;
```

Manipulating Data With SQL Statements

Data manipulation language (DML) statements query or manipulate data in existing schema objects. They enable you to:

- Add new rows of data into a table or view (`INSERT`)
- Change column values in existing rows of a table or view (`UPDATE`)
- Remove rows from tables or views (`DELETE`)

DML statements are the most frequently used SQL statements.

This section contains the following topics:

- Adding Data With the INSERT Statement on page 3-20
- Updating Data With the UPDATE Statement on page 3-20
- Deleting Data With the DELETE Statement on page 3-21

Adding Data With the INSERT Statement

You can use the SQL `INSERT` statement to add a row of data to a table. The data inserted must be valid for the datatype and size of each column of the table. See "Managing Database Objects With Object Browser" on page 2-2.

Example 3–28 shows how to use `INSERT` to add a row to the `employees` table. In the first `INSERT` statement, values are inserted into all columns in a row of the table. In the second `INSERT` statement, values are inserted only into the specified columns of the table and the remaining columns are set to `NULL`. If the those remaining columns had been specified with a `NOT NULL` constraint for the table, an error would occur. For information about constraints, see "Managing Tables" on page 2-12 and "NOT NULL Constraint" on page 2-14.

Example 3–28 Using the SQL INSERT Statement to Add Rows to a Table

```
-- the following inserts data for all the columns in a row
INSERT INTO employees VALUES
  (10, 'Enrique', 'Borges', 'enrique.borges', '555.111.2222',
   '01-AUG-05', 'AC_MGR', 9000, .1, 101, 110);

-- the following inserts data into the columns specified by name
-- NULLs are inserted in those columns not explicitly named
INSERT INTO employees (employee_id, last_name, email, hire_date, job_id, salary)
  VALUES (11, 'Doe', 'jane.doe', '31-AUG-05', 'SH_CLERK', 2400);

-- the following shows the rows that were inserted
SELECT employee_id, last_name FROM employees
  WHERE employee_id = 10 or employee_id = 11;
```

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

Updating Data With the UPDATE Statement

You can use the SQL `UPDATE` statement to update data in a row of a table. The updated data must be valid for the datatype and size of each column of the table.

Example 3–29 shows how to use `UPDATE` to update data in the `employees` table. Note the use of the multiplication operator `*` to calculate a new salary. For information about arithmetic operators, See "Using Arithmetic Operators" on page 3-13.

Example 3–29 Using the SQL UPDATE Statement to Update Data in a Table

```
SELECT salary FROM employees WHERE employee_id = 11;

-- update the salary for employee 11, multiply the salary by 105%
UPDATE employees SET salary = salary * 1.05 WHERE employee_id = 11;

-- the following should show a change in salary
SELECT salary FROM employees WHERE employee_id = 11;
```

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

Deleting Data With the DELETE Statement

With the SQL `DELETE` statement, you can delete all or specific rows in a table.

When you delete all the rows in a table, the empty table still exists. If you want to remove the entire table from the database, use the SQL `DROP` statement. See "Dropping a Table With SQL" on page 3-25.

Example 3–30 shows how to use `DELETE` to delete selected rows in the `employees` table. Note the use of the `WHERE` clause. Without that clause, all the rows would be deleted.

Example 3–30 Using the SQL DELETE Statement to Remove Rows From a Table

```
DELETE FROM employees WHERE employee_id = 10 OR employee_id = 11;

-- the following query should not find any records
SELECT * FROM employees WHERE employee_id = 10 OR employee_id = 11;
```

If you accidentally delete rows, you can restore the rows with the `ROLLBACK` statement. See "Rolling Back a Transaction" on page 3-22.

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

Using Transaction Control Statements

Transaction control statements manage the changes made by DML statements and group DML statements into transactions. They enable you to:

- Make a changes in transactions permanent (`COMMIT`)
- Undo the changes in a transaction, either since the transaction started or since a savepoint (`ROLLBACK`)

This section contains the following topics:

- Committing Transaction Changes on page 3-22
- Rolling Back a Transaction on page 3-22

Committing Transaction Changes

The SQL `COMMIT` statement saves any changes made to the database. When a `COMMIT` has been issued, all the changes since the last `COMMIT`, or since you logged on as the current user, are saved.

Note: If you are using SQL Commands, the Autocommit feature can be set to automatically commit changes after issuing SQL statements.

Example 3–31 shows how to use `COMMIT` to commit (save) changes to the `employees` table in the database.

Example 3–31 Using the SQL COMMIT Statement to Save Changes

```
-- add a row and then update the data
INSERT INTO employees (employee_id, last_name, email, hire_date, job_id, salary)
  VALUES (12, 'Doe', 'john.doe', '31-AUG-05', 'SH_CLERK', 2400);

UPDATE employees SET salary = salary*1.10 WHERE employee_id = 12;

-- commit (save) the INSERT and UPDATE changes in the database
COMMIT;
```

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

Rolling Back a Transaction

You can use the SQL `ROLLBACK` statement to rollback (undo) any changes you made to the database before a `COMMIT` was issued.

Example 3–32 shows how to use `ROLLBACK` to rollback the deletions made to the `employees` table. Note that the `ROLLBACK` was issued before a `COMMIT` was issued.

Note: If you are using SQL Commands, disable the Autocommit feature when trying this example.

Example 3–32 Using the SQL ROLLBACK Statement to Undo Changes

```
-- delete a row (record)
DELETE FROM employees WHERE last_name = 'Doe';

-- rollback the delete statement because the previous DELETE was incorrect
ROLLBACK;

-- the following is valid
SELECT * FROM employees WHERE last_name = 'Doe';
```

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

Using Data Definition Language Statements to Manage Database Objects

Data definition language (DDL) statements include `CREATE`, `ALTER`, and `DROP` to manage database objects. When managing database objects, the Object Browser page provides a Web-based user interface that can be used instead of SQL DDL statements. See "Managing Database Objects With Object Browser" on page 2-2.

In this guide, some basic SQL DDL statements are used in the code examples and a brief description of some DDL statements are discussed in this section.

This section contains the following topics:

- Creating a Table With SQL on page 3-23
- Adding, Altering, and Dropping a Table Column With SQL on page 3-24
- Creating and Altering a Constraint With SQL on page 3-24
- Renaming a Table With SQL on page 3-25
- Dropping a Table With SQL on page 3-25
- Creating, Altering, and Dropping an Index With SQL on page 3-25
- Creating and Dropping a View With SQL on page 3-26
- Creating and Dropping a Sequence With SQL on page 3-26
- Creating and Dropping a Synonym With SQL on page 3-27

Creating a Table With SQL

To create a database object, such as a table, use the SQL `CREATE` statement as shown in Example 3–33. When you create a table, you need to provide datatypes for each column. For more information about tables, see "Managing Tables" on page 2-12.

Example 3–33   Creating a Simple Table Using SQL

```
-- create a simple table for keeping track of birthdays
CREATE TABLE my_birthdays
  ( first_name     VARCHAR2(20),
    last_name      VARCHAR2(25),
    bday_date      DATE
  );
```

Optionally, you can provide `NOT NULL` constraints, as shown in Example 3–34. The `NOT NULL` constraint is discussed in "NOT NULL Constraint" on page 2-14.

Example 3–34   Creating a Table With NOT NULL Constraints Using SQL

```
-- create a table with NOT NULL constraints in the HR schema
CREATE TABLE personal_info (
    employee_id          NUMBER(6,0) NOT NULL,
    birth_date           DATE NOT NULL,
    social_security_id   VARCHAR2(12) NOT NULL,
    marital_status       VARCHAR2(10),
    dependents_claimed   NUMBER(2,0) DEFAULT 1,
    contact_name         VARCHAR2(45) NOT NULL,
    contact_phone        VARCHAR2(20) NOT NULL,
    contact_address      VARCHAR2(80) NOT NULL
);
```

For information about creating a table with the Object Browser page, see "Creating a Table" on page 2-16.

Adding, Altering, and Dropping a Table Column With SQL

To alter a database object, such as a table, use the SQL `ALTER` statement, as shown in Example 3–35.

Example 3–35 Adding, Altering, and Dropping a Table Column Using SQL

```
-- add a new column
ALTER TABLE personal_info ADD
(contact_email VARCHAR2(30) NULL);

-- modify a column
ALTER TABLE personal_info MODIFY
(contact_email VARCHAR2(40) NOT NULL);

-- drop a column
ALTER TABLE personal_info DROP COLUMN
contact_address;
```

For information about adding, modifying, and dropping a table column with the Object Browser page, see "Adding a Column To a Table" on page 2-18, "Modifying a Column In a Table" on page 2-18, and "Dropping a Column From a Table" on page 2-19.

Creating and Altering a Constraint With SQL

To add, alter, or drop a constraint on a table, use the SQL `ALTER` statement, as shown in Example 3–36. For information about primary key, foreign key, unique, and check constraints, see "Ensuring Data Integrity in Tables With Constraints" on page 2-13.

Example 3–36 Creating, Altering, and Dropping Constraints Using SQL

```
-- add a primary key constraint
ALTER TABLE personal_info
  ADD CONSTRAINT personal_info_pkey
  PRIMARY KEY (employee_id);

-- add a foreign key constraint
ALTER TABLE personal_info
  ADD CONSTRAINT personal_info_fkey
  FOREIGN KEY (employee_id) REFERENCES employees (employee_id)
  ON DELETE CASCADE;

-- add a unique constraint
ALTER TABLE personal_info
  ADD CONSTRAINT personal_info_unique_con
  UNIQUE (social_security_id);

-- add a check constraint
ALTER TABLE personal_info
  ADD CONSTRAINT personal_info_check_con
  CHECK ( dependents_claimed > 0);

-- disable a constraint
ALTER TABLE personal_info
  DISABLE CONSTRAINT personal_info_check_con;

-- enable a constraint
ALTER TABLE personal_info
  ENABLE CONSTRAINT personal_info_check_con;
```

```
-- drop a constraint
ALTER TABLE personal_info
  DROP CONSTRAINT personal_info_check_con;
```

For information about adding a constraint with the Object Browser page, see "Adding a Primary Key Constraint" on page 2-21, "Adding a Foreign Key Constraint" on page 2-22, "Adding a Unique Constraint" on page 2-20, and "Adding a Check Constraint" on page 2-19.

Renaming a Table With SQL

To rename a database object, such as a table, use the SQL `ALTER` statement, as shown in Example 3–37.

Example 3–37 Renaming a Table Using SQL

```
-- rename the my_birthdays table
ALTER TABLE my_birthdays RENAME to birthdays;
```

Dropping a Table With SQL

To drop (remove completely) a table from the database, use the SQL `DROP` statement, as shown in Example 3–38. Be careful when using the `DROP` statement to remove database objects.

If you want to delete the rows in the table and keep the table, use the `DELETE` statement. See "Deleting Data With the DELETE Statement" on page 3-21.

Example 3–38 Dropping a Table Using SQL

```
-- drop tables from the database
-- use caution when use the DROP statement!
DROP TABLE birthdays;
DROP TABLE personal_info;
```

Creating, Altering, and Dropping an Index With SQL

To create, modify, or drop an index, use the SQL `CREATE`, `ALTER`, or `DROP INDEX` statement, as shown in Example 3–39. For more information about indexes, see "Managing Indexes" on page 2-26.

Example 3–39 Creating, Modifying, and Dropping an Index Using SQL

```
-- create an index on a single column to make queries faster on that column
CREATE INDEX emp_hiredate_idx ON employees (hire_date);

-- rename the index
ALTER INDEX emp_hiredate_idx
  RENAME TO emp_hire_date_idx;

-- drop the index
DROP INDEX emp_hire_date_idx;

-- create an index on two columns to make queries faster on the first column
-- or both columns
CREATE INDEX emp_mgr_id_ix ON employees (employee_id, manager_id);
DROP INDEX emp_mgr_id_ix;
```

```
-- a function-based index precalculates the result and speeds up queries that
-- use the function for searching or sorting, in this case UPPER(last_name)
CREATE INDEX emp_upper_last_name_ix ON employees (UPPER(last_name));
DROP INDEX emp_upper_last_name_ix;
```

For information about creating an index with the Object Browser page, see "Creating an Index" on page 2-29.

Creating and Dropping a View With SQL

To create a database object, such as a view, use the SQL `CREATE` statement as shown in Example 3–40. For more information about views, see "Managing Views" on page 2-31.

Example 3–40 Creating a View Using SQL

```
-- create a view to display data from departments and employees
CREATE OR REPLACE VIEW my_emp_view AS
SELECT d.department_id, d.department_name,
  e.employee_id, e.first_name, e.last_name
  FROM employees e
  JOIN departments d ON d.manager_id = e.employee_id;
```

Example 3–41 shows how to drop the view that you previously created.

Example 3–41 Dropping a View Using SQL

```
-- drop the view with the DROP VIEW statement
DROP VIEW my_emp_view;
```

For information about creating and dropping a view with the Object Browser page, see "Creating a View" on page 2-32 and "Dropping a View" on page 2-33. For an additional example of creating a view, see Example 6–3 on page 6-12.

Creating and Dropping a Sequence With SQL

A sequence is a database object that generates unique sequential values, often used for primary and unique keys. You can refer to sequence values in SQL statements with the `CURRVAL` and `NEXTVAL` pseudocolumns.

To generate a sequence number, you call the sequence using the `CURRVAL` or `NEXTVAL` keywords. You must qualify `CURRVAL` and `NEXTVAL` with the name of the sequence, such as `employees_seq.CURRVAL` or `employees_seq.NEXTVAL`. Before you use `CURRVAL` for a sequence in your session, you must first initialize the sequence with `NEXTVAL`.

Example 3–42 shows how to create a sequence that can be used with the `employees` table. The sequence can also be used with other tables. For more information about sequences, see "Managing Sequences" on page 2-34.

Example 3–42 Creating a Sequence Using SQL

```
-- create a new sequence to use with the employees table
-- this sequence starts at 1000 and increments by 1
CREATE SEQUENCE new_employees_seq START WITH 1000 INCREMENT BY 1;

-- to use the sequence, first initialize the sequence with NEXTVAL
SELECT new_employees_seq.NEXTVAL FROM DUAL;

-- after initializing the sequence, use CURRVAL as the next value in the sequence
INSERT INTO employees VALUES
```

```
  (new_employees_seq.CURRVAL, 'Pilar', 'Valdivia', 'pilar.valdivia',
  '555.111.3333', '01-SEP-05', 'AC_MGR', 9100, .1, 101, 110);

-- query the employees table to check the current value of the sequence
-- which was inserted used as employee_id in the previous INSERT statement
SELECT employee_id, last_name FROM employees WHERE last_name = 'Valdivia';
```

Example 3–43 shows how to drop the sequence that you previously created.

Example 3–43 Dropping a Sequence Using SQL

```
-- drop the sequence
DROP SEQUENCE new_employees_seq;
```

For information about creating and dropping a sequence with the Object Browser page, see "Creating a Sequence" on page 2-34 and "Dropping a Sequence" on page 2-35.

Creating and Dropping a Synonym With SQL

Example 3–44 shows how to create a synonym that is an alias for the `employees` table. For more information about synonyms, see "Managing Synonyms" on page 2-35.

Example 3–44 Creating a Synonym Using SQL

```
-- create a synonym for the employees table
CREATE SYNONYM emps for HR.employees;

-- query the employees table using the emps synonym
SELECT employee_id, last_name FROM emps WHERE employee_id < 105;
```

Example 3–45 show how to drop a synonym.

Example 3–45 Dropping a Synonym Using SQL

```
-- drop the synonym
DROP SYNONYM emps;
```

For information about creating and dropping a synonym with the Object Browser page, see "Creating a Synonym" on page 2-36 and "Dropping a Synonym" on page 2-36.

4 Using PL/SQL

This section discusses the PL/SQL language, which can be use to develop applications for Oracle Database Express Edition.

This section contains the following topics:

See Also:

- *Oracle Database PL/SQL User's Guide and Reference* for detailed information about PL/SQL
- *Oracle Database PL/SQL Packages and Types Reference* for information about packages supplied by Oracle
- *Oracle Database Application Developer's Guide - Fundamentals* for information about dynamic SQL
- *Oracle Database Application Developer's Guide - Fundamentals* for information about using PL/SQL to develop Web applications

Overview of PL/SQL

PL/SQL is an Oracle's procedural language extension to SQL. It is a server-side, stored procedural language that is easy-to-use, seamless with SQL, portable, and secure.

PL/SQL enables you to mix SQL statements with procedural constructs. With PL/SQL, you can create and run PL/SQL program units such as procedures, functions, and packages. PL/SQL program units generally are categorized as anonymous blocks, stored functions, stored procedures, and packages.

The following can be constructed with the PL/SQL language:

- Anonymous block

  An anonymous block is a PL/SQL block that appears in your application and is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear. A PL/SQL block groups related declarations and statements. Because these blocks are not stored in the database, they are generally for one-time use.

- Stored or standalone procedure and function

A stored procedure or function is a PL/SQL block that Oracle Database XE stores in the database and can be called by name from an application. Functions are different than procedures in that functions return a value when executed. When you create a stored procedure or function, Oracle Database XE parses the procedure or function, and stores its parsed representation in the database. See Chapter 5, "Using Procedures, Functions, and Packages".

- Package

  A package is a group of procedures, functions, and variable definitions that Oracle Database XE stores in the database. Procedures, functions, and variables in packages can be called from other packages, procedures, or functions. See Chapter 5, "Using Procedures, Functions, and Packages".

- Trigger

  A database trigger is a stored procedure associated with a database table, view, or event. The trigger can be called after the event, to record it, or take some follow-up action. The trigger can be called before the event, to prevent erroneous operations or fix new data so that it conforms to business rules. See Chapter 6, "Using Triggers".

Entering and Running PL/SQL Code

You can enter and run PL/SQL code from the SQL Commands page, Script Editor page, or SQL Command Line (SQL*Plus).

Using the SQL Commands page is described in this section. The SQL Commands page is a simpler interface and easier to use.

Both SQL Commands and Script Editor pages enable you to save your SQL statements as a script file in a database repository for future use. You can run multiple SQL statements in the Script Editor page. Script Editor also enables you to download the script to the local file system. For information about using the Script Editor page, see "Running SQL Statements in the Script Editor Page" on page 3-3.

You can create a text file of the PL/SQL code with the Script Editor page or a text editor to run as a SQL script from SQL Command Line. Using a script makes correcting mistakes easier because you only need to make the necessary updates to correct the problem, rather than entering again all the PL/SQL code at the SQL Command Line prompt. For information about using SQL Command Line and running SQL scripts from SQL Command Line, see Appendix A, "Using SQL Command Line".

This section contains the following topic:

- Running PL/SQL Code in the SQL Commands Page on page 4-2

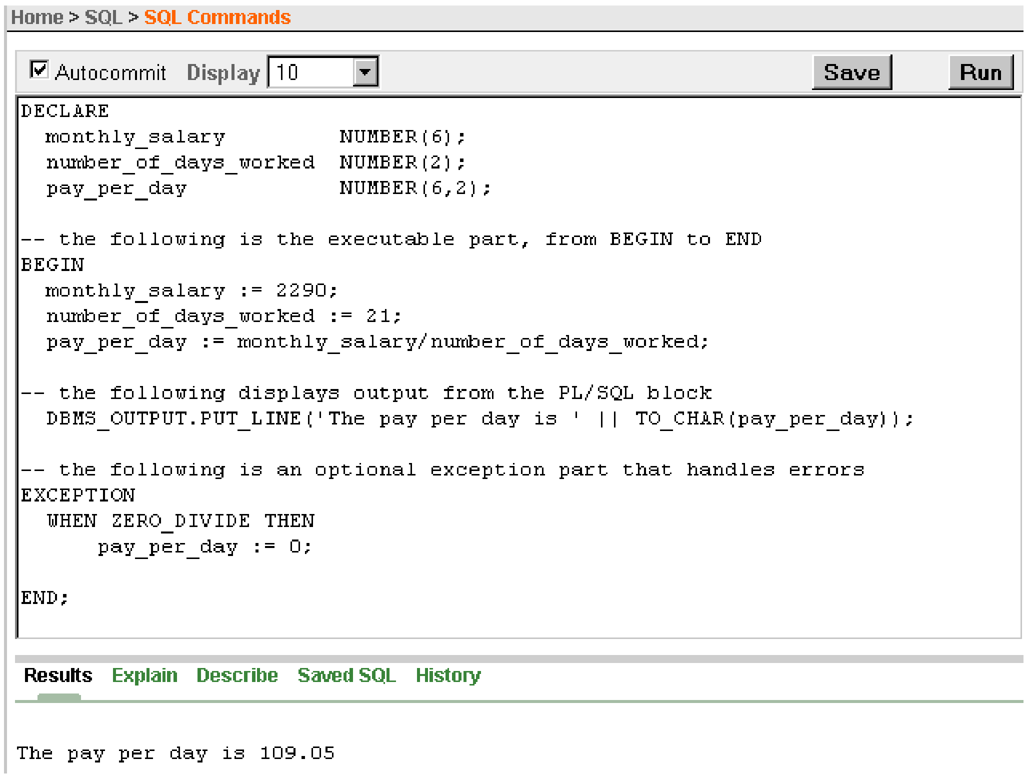
Running PL/SQL Code in the SQL Commands Page

To enter and run PL/SQL code in the SQL Commands page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the home page, click the **SQL** icon to display the SQL page.
3. Click the **SQL Commands** icon to display the SQL Commands page.
4. On the SQL Commands page, enter the PL/SQL code in Example 4–1 on page 4-4. Note that some of the lines of code are terminated with a semi colon (;) and the

entire code unit is terminated with a slash (/). The slash is required when running the PL/SQL in a SQL script or at the SQL Command Line prompt, but it is optional on the SQL Commands page.

5. Click the **Run** button to run the PL/SQL code. If necessary, select (highlight) only the PL/SQL code block before clicking the **Run** button. Any comments outside the PL/SQL code block are not legal in the SQL Commands page.



```
DECLARE
  monthly_salary         NUMBER(6);
  number_of_days_worked  NUMBER(2);
  pay_per_day            NUMBER(6,2);

-- the following is the executable part, from BEGIN to END
BEGIN
  monthly_salary := 2290;
  number_of_days_worked := 21;
  pay_per_day := monthly_salary/number_of_days_worked;

-- the following displays output from the PL/SQL block
  DBMS_OUTPUT.PUT_LINE('The pay per day is ' || TO_CHAR(pay_per_day));

-- the following is an optional exception part that handles errors
EXCEPTION
  WHEN ZERO_DIVIDE THEN
      pay_per_day := 0;

END;
```

```
The pay per day is 109.05
```

6. If you want to save the PL/SQL code for future use, click the **Save** button.

7. In the **Name** field, enter a name for the saved PL/SQL code. You can also enter an optional description. Click the **Save** button to save the SQL.

8. To access saved PL/SQL code, click the **Saved SQL** tab, and select the name of the saved PL/SQL code that you want to access.

> **See Also:** *Oracle Database Express Edition Application Express User's Guide* for detailed information about using SQL Commands

Using the Main Features of PL/SQL

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages. You can control program flow with statements, such as `IF` and `LOOP`. As with other procedural programming languages, you can declare variables, define procedures and functions, and trap run time errors.

PL/SQL lets you break complex problems down into understandable procedural code, and reuse this code across multiple applications. When a problem can be solved through plain SQL, you can issue SQL statements directly inside your PL/SQL programs, without learning new APIs. PL/SQL datatypes correspond with SQL column types, enabling you to interchange PL/SQL variables with data inside a table.

This section contains the following topics:

Using the PL/SQL Block Structure

As Example 4–1 shows, a PL/SQL block has three basic parts: a declarative part (DECLARE), an executable part (BEGIN ... END), and an exception-handling (EXCEPTION) part that handles error conditions. For a discussion about exception handling, see "Handling PL/SQL Errors" on page 4-28.

Only the executable part is required. The optional declarative part is written first, where you define types, variables, and similar items. These items are manipulated in the executable part. Errors that occur during execution can be dealt with in the exception-handling part.

Note the comments that are added to the PL/SQL code. See "Using Comments" on page 4-6. Also, note the use of DBMS_OUTPUT.PUT_LINE to display output. See "Inputting and Outputting Data with PL/SQL" on page 4-5.

Example 4–1 Using a Simple PL/SQL Block

```
-- the following is an optional declarative part
DECLARE
  monthly_salary         NUMBER(6);
  number_of_days_worked  NUMBER(2);
  pay_per_day            NUMBER(6,2);

-- the following is the executable part, from BEGIN to END
BEGIN
  monthly_salary := 2290;
  number_of_days_worked := 21;
  pay_per_day := monthly_salary/number_of_days_worked;
```

```
-- the following displays output from the PL/SQL block
  DBMS_OUTPUT.PUT_LINE('The pay per day is ' || TO_CHAR(pay_per_day));

-- the following is an optional exception part that handles errors
EXCEPTION
  WHEN ZERO_DIVIDE THEN
      pay_per_day := 0;

END;
/
```

For another example of a PL/SQL block structure, see Example 4–13 on page 4-11.

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information about PL/SQL language elements

Inputting and Outputting Data with PL/SQL

Most PL/SQL input and output is through SQL statements, to store data in database tables or to query those tables. All other PL/SQL I/O is done through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures such as PUT_LINE. To see the result outside of PL/SQL requires another program, such as the SQL Commands page or SQL Command Line (SQL*Plus), to read and display the data passed to DBMS_OUTPUT.

The SQL Commands page is configured to display output with DBMS_OUTPUT. SQL Command Line does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON. For information about SQL Command Line SET command, see "SQL Command Line SET Commands" on page A-4.

Example 4–2 show the use of DBMS_OUTPUT.PUTLINE. Note the use of SET SERVEROUTPUT ON to enable output.

Example 4–2 Using DBMS_OUTPUT.PUT_LINE to Display PL/SQL Output

```
-- enable SERVEROUTPUT in SQL Command Line (SQL*Plus) to display output with
-- DBMS_OUTPUT.PUT_LINE, this enables SERVEROUTPUT for this SQL*Plus session only
SET SERVEROUTPUT ON

DECLARE
  answer  VARCHAR2(20); -- declare a variable
BEGIN
-- assign a value to a variable
  answer := 'Maybe';
-- use PUT_LINE to display data from the PL/SQL block
  DBMS_OUTPUT.PUT_LINE( 'The answer is: ' || answer );
END;
/
```

The DBMS_OUTPUT package is a predefined Oracle package. For information about Oracle supplied packages, see "Oracle Provided Packages" on page 5-23.

> **See Also:**
>
> - *SQL*Plus User's Guide and Reference* for information SQL*Plus commands
> - *Oracle Database PL/SQL Packages and Types Reference* for information about Oracle supplied packages

Using Comments

The PL/SQL compiler ignores comments, but you should not. Adding comments to your program improves readability and helps others understand your code. Generally, you use comments to describe the purpose and use of each code segment. PL/SQL supports single-line and multiple-line comment styles.

Single-line comments begin with a double hyphen (`--`) anywhere on a line and extend to the end of the line. Multiple-line comments begin with a slash and an asterisk (`/*`), end with an asterisk and a slash (`*/`), and can span multiple lines. See Example 4–3.

Example 4–3 Using Comments in PL/SQL

```
DECLARE  -- Declare variables here.
  monthly_salary         NUMBER(6);  -- This is the monthly salary.
  number_of_days_worked  NUMBER(2);  -- This is the days in one month.
  pay_per_day            NUMBER(6,2); -- Calculate this value.
BEGIN
-- First assign values to the variables.
  monthly_salary := 2290;
  number_of_days_worked := 21;

-- Now calculate the value on the following line.
  pay_per_day := monthly_salary/number_of_days_worked;

-- the following displays output from the PL/SQL block
  DBMS_OUTPUT.PUT_LINE('The pay per day is ' || TO_CHAR(pay_per_day));

EXCEPTION
/* This is a simple example of an exeception handler to trap division by zero.
   In actual practice, it would be best to check whether a variable is
   zero before using it as a divisor. */
  WHEN ZERO_DIVIDE THEN
      pay_per_day := 0; -- set to 0 if divisor equals 0
END;
/
```

While testing or debugging a program, you might want to disable a line of code. The following example shows how you can disable a single line by making it a comment:

```
-- pay_per_day := monthly_salary/number_of_days_worked;
```

You can use multiple-line comment delimiters to comment out large sections of code.

Declaring Variables and Constants

Variables can have any SQL datatype, such as `VARCHAR2`, `DATE`, or `NUMBER`, or a PL/SQL-only datatype, such as a `BOOLEAN` or `PLS_INTEGER`. You can also declare nested tables, variable-size arrays (varrays for short), and records using the `TABLE`, `VARRAY`, and `RECORD` composite datatypes. See "Working With PL/SQL Data Structures" on page 4-24.

Declaring a constant is similar to declaring a variable except that you must add the `CONSTANT` keyword and immediately assign a value to the constant. No further assignments to the constant are allowed. For an example, see the `avg_days_worked_month` constant in Example 4–4.

For example, assume that you want to declare variables for employee data, such as `employee_id` to hold 6-digit numbers and `active_employee` to hold the Boolean value `TRUE` or `FALSE`. You declare these and related employee variables and constants, as shown in Example 4–4.

Note that there is a semi colon (;) at the end of each line in the declaration section. Also, note the use of the `NULL` statement that enables you to run and test the PL/SQL block.

You can choose any naming convention for variables that is appropriate for your application, but the names must be valid PL/SQL identifiers. See "Using Identifiers in PL/SQL" on page 4-7.

Example 4–4 Declaring Variables in PL/SQL

```
DECLARE -- declare the variables in this section
  last_name             VARCHAR2(30);
  first_name            VARCHAR2(25);
  employee_id           NUMBER(6);
  active_employee       BOOLEAN;
  monthly_salary        NUMBER(6);
  number_of_days_worked  NUMBER(2);
  pay_per_day           NUMBER(6,2);
  avg_days_worked_month  CONSTANT NUMBER(2) := 21; -- a constant variable
BEGIN
  NULL; -- NULL statement does nothing, allows this block to executed and tested
END;
/
```

See Also: *Oracle Database PL/SQL User's Guide and Reference* for information about datatypes used with PL/SQL, including the PL/SQL `BOOLEAN` and `PLS_INTEGER` datatypes

Using Identifiers in PL/SQL

You use identifiers to name PL/SQL program items and units, such as constants, variables, exceptions, and subprograms. An identifier consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs. Characters such as ampersands (&), hyphens (-), slashes (/), and spaces ( ) are not allowed.

You can use uppercase, lowercase, or mixed case to write identifiers. PL/SQL is not case-sensitive except within string and character literals. Every character, including dollar signs, underscores, and number signs, is significant. If the only difference between identifiers is the case of corresponding letters, PL/SQL considers the identifiers the same.

The declaration section in Example 4–5 show some PL/SQL identifiers. You can see additional examples of identifiers for variable names in Example 4–3 on page 4-6 and Example 4–4 on page 4-7.

Example 4–5 Using Identifiers for Variables in PL/SQL

```
DECLARE
  lastname          VARCHAR2(30); -- valid identifier
  last_name         VARCHAR2(30); -- valid identifier, _ allowed
  last$name         VARCHAR2(30); -- valid identifier, $ allowed
  last#name         VARCHAR2(30); -- valid identifier, # allowed
--  last-name  is invalid, hypen not allowed
--  last/name  is invalid, slash not allowed
--  last name  is invalid, space not allowed
--  LASTNAME is invalid, same as lastname and LastName
--  LastName is invalid, same as lastname and LASTNAME
BEGIN
```

```
  NULL; -- NULL statement does nothing, allows this block to executed and tested
END;
/
```

The size of an identifier cannot exceed 30 characters. Identifiers should be descriptive. When possible, avoid obscure names such as `cpm`. Instead, use meaningful names such as `cost_per_million`. You can use prefixes for more clarification. For example, you could begin each variable name with the *`var_`* and each constant name with *`con_`*.

Some identifiers, called reserved words or keywords, have a special syntactic meaning to PL/SQL. For example, the words `BEGIN` and `END` are reserved. Often, reserved words and keywords are written in upper case for readability. Neither reserved words or keywords should be used as identifiers and the use can cause compilation errors. For a list of PL/SQL reserved words and keywords, see Appendix B, "Reserved Words".

Assigning Values to a Variable With the Assignment Operator

You can assign values to a variable in several ways. One way uses the assignment operator (`:=`), a colon followed by an equal sign, as shown in Example 4–6. You place the variable to the left of the operator and an expression, including function calls, to the right. Note that you can assign a value to a variable when it is declared.

Example 4–6   Assigning Values to Variables With the PL/SQL Assignment Operator

```
DECLARE  -- declare and assiging variables
   wages          NUMBER(6,2);
   hours_worked   NUMBER := 40;
   hourly_salary  NUMBER := 22.50;
   bonus          NUMBER := 150;
   country        VARCHAR2(128);
   counter        NUMBER := 0;
   done           BOOLEAN := FALSE;
   valid_id       BOOLEAN;
BEGIN
   wages := (hours_worked * hourly_salary) + bonus;  -- compute wages
   country := 'France'; -- assign a string literal
   country := UPPER('Canada'); -- assign an uppercase string literal
   done := (counter > 100); -- assign a BOOLEAN, in this case FALSE
   valid_id := TRUE; -- assign a BOOLEAN
END;
/
```

Using Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, `147` is a numeric literal, and `FALSE` is a Boolean literal.

Numeric Literals

Two kinds of numeric literals can be used in arithmetic expressions: integer and real. An integer literal is an optionally signed whole number without a decimal point, such as `+6`. A real literal is an optionally signed whole or fractional number with a decimal point, such as `-3.14159`. PL/SQL considers a number such as `25.` to be real, even though it has an integral value.

Numeric literals cannot contain dollar signs or commas, but can be written using scientific notation. Add an `E` (or `e`) after the base number, followed by an optionally signed integer, for example `-9.5e-3`. The `E` (or e) represents the base number ten and the following integer represents the exponent.

Example 4–7 shows some examples of numeric literals.

Example 4–7 Using Numeric Literals in PL/SQL

```
DECLARE  -- declare and assign variables
  number1 PLS_INTEGER := 32000;  -- numeric literal
  number2 NUMBER(8,3);
BEGIN
  number2 := 3.125346e3;  -- numeric literal
  number2 := -8300.00;  -- numeric literal
  number2 := -14;  -- numeric literal
END;
/
```

Character Literals

A character literal is an individual character enclosed by single quotation marks (apostrophes), such as `'('` or `'7'`. Character literals include all the printable characters in the PL/SQL character set: letters, numbers, spaces, and special symbols.

PL/SQL is case-sensitive within character literals. For example, PL/SQL considers the character literals `'Z'` and `'z'` to be different. The character literals `'0'`...`'9'` are not equivalent to integer literals, but can be used in arithmetic expressions because they are implicitly convertible to integers.

Example 4–8 shows some examples of character literals.

Example 4–8 Using Character Literals in PL/SQL

```
DECLARE  -- declare and assign variables
  char1   VARCHAR2(1) := 'x'; -- character literal
  char2   VARCHAR2(1);
BEGIN
  char2 := '5'; -- character literal
END;
/
```

String Literals

A character value can be represented by an identifier or explicitly written as a string literal, which is a sequence of zero or more characters enclosed by single quotation marks, such as `'Hello, world!'` and `'$1,000,000'`.

PL/SQL is case-sensitive within string literals. For example, PL/SQL considers the string literals `'baker'` and `'Baker'` to be different:

To represent an apostrophe within a string, you can use two single quotation marks (`''`), which is not the same as a quotation mark (`"`). You can also use the quote-delimiter mechanism, which enables you to specify `q` or `Q` followed by a single quotation mark and then another character to be used as the quotation mark delimiter. See "Using Character Literals in SQL Statements" on page 3-7.

Example 4–9 shows some examples of string literals.

Example 4–9 Using String Literals in PL/SQL

```
DECLARE  -- declare and assign variables
  string1   VARCHAR2(1000);
  string2   VARCHAR2(32767);
BEGIN
  string1 := '555-111-2323';
 -- the following needs two single quotation marks to represent one in the string
  string2 := 'Here''s an example of two single quotation marks used in a string.';
END;
/
```

BOOLEAN Literals

BOOLEAN literals are the predefined values: TRUE, FALSE, and NULL. NULL is a missing, unknown, or inapplicable value. BOOLEAN literals are values, not strings.

Example 4–10 shows some examples of BOOLEAN literals.

Example 4–10   Using BOOLEAN Literals in PL/SQL

```
DECLARE  -- declare and assign variables
  finished      BOOLEAN := TRUE; -- BOOLEAN literal
  complete      BOOLEAN; -- BOOLEAN literal
  true_or_false BOOLEAN;
BEGIN
  finished := FALSE;  -- BOOLEAN literal set to FALSE
  complete := NULL; -- BOOLEAN literal with unknown value
  true_or_false := (3 = 4); -- BOOLEAN literal set to FALSE
  true_or_false := (3 < 4); -- BOOLEAN literal set to TRUE
END;
/
```

Date-time Literals

Date-time literals have various formats depending on the date-time datatype used, such as '14-SEP-05' or '14-SEP-05 09:24:04 AM'.

Example 4–11 shows some examples of date-time literals.

Example 4–11   Using Date-time Literals in PL/SQL

```
DECLARE  -- declare and assign variables
  date1   DATE := '11-AUG-2005'; -- DATE literal
  time1   TIMESTAMP;
  time2   TIMESTAMP WITH TIME ZONE;
BEGIN
  time1 := '11-AUG-2005 11:01:01 PM'; -- TIMESTAMP literal
  time2 := '11-AUG-2005 09:26:56.66 PM +02:00'; -- TIMESTAMP WITH TIME ZONE
END;
/
```

See Also:

- *Oracle Database SQL Reference* for information about the syntax for literals and the date and time types
- *Oracle Database Application Developer's Guide - Fundamentals* for examples of performing date and time arithmetic
- *Oracle Database PL/SQL User's Guide and Reference* for information about using literals with PL/SQL

Declaring Variables With the DEFAULT Keyword or NOT NULL Constraint

You can use the `DEFAULT` keyword instead of the assignment operator to initialize variables when they are declared. Use `DEFAULT` for variables that have a typical value. Use the assignment operator for variables (such as counters and accumulators) that have no typical value. You can also use `DEFAULT` to initialize subprogram parameters, cursor parameters, and fields in a user-defined record.

In addition to assigning an initial value, declarations can impose the `NOT NULL` constraint so that assigning a `NULL` causes an error. The `NOT NULL` constraint must be followed by an initialization clause.

In Example 4–12 the declaration for the `avg_days_worked_month` variable uses the `DEFAULT` to assign a value of 21 and the declarations for the `active_employee` and `monthly_salary` variables use the `NOT NULL` constraint.

Example 4–12 Using DEFAULT and NOT NULL in PL/SQL

```
DECLARE  -- declare and assign variables
  last_name              VARCHAR2(30);
  first_name             VARCHAR2(25);
  employee_id            NUMBER(6);
  active_employee        BOOLEAN NOT NULL := TRUE;  -- value cannot be NULL
  monthly_salary         NUMBER(6) NOT NULL := 2000; -- value cannot be NULL
  number_of_days_worked  NUMBER(2);
  pay_per_day            NUMBER(6,2);
  employee_count         NUMBER(6) := 0;
  avg_days_worked_month  NUMBER(2) DEFAULT 21;  -- assign a default value
BEGIN
  NULL; -- NULL statement does nothing, allows this block to executed and tested
END;
/
```

Assigning Values to a Variable With the PL/SQL SELECT INTO Statement

Another way to assign values to a variable is by selecting (or fetching) database values into it. With the PL/SQL `SELECT INTO` statement, you can retrieve data from one row in a table. In Example 4–13, 10 percent of the salary of an employee is selected into the `bonus` variable. Now, you can use the `bonus` variable in another computation, or insert its value into a database table.

In the example, the `DBMS_OUTPUT.PUT_LINE` procedure is used to display output from the PL/SQL program. For more information, see "Inputting and Outputting Data with PL/SQL" on page 4-5.

Example 4–13 Assigning Values to Variables Using PL/SQL SELECT INTO

```
DECLARE -- declare and assign values
  bonus_rate CONSTANT NUMBER(2,3) := 0.05;
  bonus      NUMBER(8,2);
  emp_id     NUMBER(6) := 120;  -- assign a test value for employee ID
BEGIN
-- retreive a salary from the employees table, then calculate the bonus and
-- assign the value to the bonus variable
  SELECT salary * bonus_rate INTO bonus FROM employees
    WHERE employee_id = emp_id;
-- display the employee_id, bonus amount, and bonus rate
    DBMS_OUTPUT.PUT_LINE ( 'Employee: ' || TO_CHAR(emp_id)
      || ' Bonus: ' || TO_CHAR(bonus) || ' Bonus Rate: ' || TO_CHAR(bonus_rate));
END;
```

```
/
```

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information about `SELECT INTO` syntax

Using %TYPE and %ROWTYPE Attributes to Declare Identical Datatypes

As part of the declaration for each PL/SQL variable, you declare its datatype. Usually, this datatype is one of the types shared between PL/SQL and SQL, such as `NUMBER` or `VARCHAR2`. For easier code maintenance that interacts with the database, you can also use the special qualifiers `%TYPE` and `%ROWTYPE` to declare variables that hold table columns or table rows.

This section contains the following topics:

- Using the %TYPE Attribute to Declare Variables on page 4-12
- Using the %ROWTYPE Attribute to Declare Variables on page 4-12

Using the %TYPE Attribute to Declare Variables

The `%TYPE` attribute provides the datatype of a variable or table column. This is particularly useful when declaring variables that will hold values of a table column. For example, suppose you want to declare variables as the same datatype as the `employee_id` and `last_name` columns in `employees` table. To declare variables named `empid` and `emplname` that have the same datatype as the table columns, use dot notation and the `%TYPE` attribute. See Example 4–14.

Example 4–14   Using %TYPE With Table Columns in PL/SQL

```
DECLARE -- declare variables using %TYPE attribute
   empid    employees.employee_id%TYPE;  -- employee_id datatype is NUMBER(6)
   emplname employees.last_name%TYPE;  -- last_name datatype is VARCHAR2(25)
BEGIN
   empid    := 100301;  -- this is OK because it fits in NUMBER(6)
--   empid  := 3018907;  -- this is too large and will cause an overflow
   emplname := 'Patel'; --  this is OK because it fits in VARCHAR2(25)
   DBMS_OUTPUT.PUT_LINE('Employee ID: ' || empid);  -- display data
   DBMS_OUTPUT.PUT_LINE('Employee name: ' || emplname); -- display data
END;
/
```

Declaring variables with the `%TYPE` attribute has two advantages. First, you do not need to know the exact datatype of the table columns. Second, if you change the database definition of columns, such as `employee_id` or `last_name`, the datatypes of `empid` and `emplname` in Example 4–14 change accordingly at run time.

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information about the `%TYPE` attribute

Using the %ROWTYPE Attribute to Declare Variables

For easier maintenance of code that interacts with the database, you can use the `%ROWTYPE` attribute to declare a variable that represents a row in a table. A PL/SQL record is the datatype that stores the same information as a row in a table.

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. For information about records, see "Using Record Types" on page 4-24.

Columns in a row and corresponding fields in a record have the same names and datatypes. In Example 4–15, you declare a record named `emp_rec`. Its fields have the same names and datatypes as the columns in the `employees` table. You use dot notation to reference fields, such as `emp_rec.last_name`.

In Example 4–15, the `SELECT` statement is used to store row information from the `employees` table into the `emp_rec` record. When you run the `SELECT INTO` statement, the value in the `first_name` column of the `employees` table is assigned to the `first_name` field of `emp_rec`; the value in the `last_name` column is assigned to the `last_name` field of `emp_rec`; and so on.

Example 4–15 Using %ROWTYPE with a PL/SQL Record

```
DECLARE -- declare variables
-- declare record variable that represents a row fetched from the employees table
   emp_rec employees%ROWTYPE; -- declare variable with %ROWTYPE attribute
BEGIN
  SELECT * INTO emp_rec FROM EMPLOYEES WHERE employee_id = 120; -- retrieve record
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || emp_rec.first_name || ' '
                       || emp_rec.last_name); -- display
END;
/
```

Declaring variables with the `%ROWTYPE` attribute has several advantages. First, you do not need to know the exact datatype of the table columns. Second, if you change the database definition of any of the table columns, the datatypes associated with the `%ROWTYPE` declaration change accordingly at run time.

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information about the `%ROWTYPE` attribute

Using PL/SQL Control Structures

Control structures are the most important PL/SQL extension to SQL. Not only does PL/SQL let you manipulate Oracle data, it lets you process the data using conditional, iterative, and sequential flow-of-control statements such as `IF-THEN-ELSE`, `CASE`, `FOR-LOOP`, `WHILE-LOOP`, `EXIT-WHEN`, and `GOTO`.

This section contains the following topics:

- Conditional Control With IF-THEN on page 4-13
- Conditional Control With the CASE Statement on page 4-14
- Iterative Control With LOOPs on page 4-15
- Sequential Control With GOTO on page 4-17

Conditional Control With IF-THEN

Often, it is necessary to take alternative actions depending on circumstances. The `IF-THEN` statement lets you run a sequence of statements conditionally. The forms of the statement can be `IF-THEN`, `IF-THEN-ELSE`, or `IF-THEN-ELSEIF-ELSE`. The `IF` clause checks a condition; the `THEN` clause defines what to do if the condition is true; and the `ELSE` clause defines what to do if the condition is false or null. Example 4–16 shows a simple use of the `IF-THEN` statement.

Example 4–16 Using a Simple IF-THEN Statement in PL/SQL

```
DECLARE
  sal         NUMBER(8,2);
  bonus       NUMBER(6,2);
  hiredate    DATE;
  empid       NUMBER(6) := 128; -- use employee 120 for testing
BEGIN
-- retrieve the salary and the date that employee was hired, the date is checked
-- to calculate the amount of the bonus for the employee
  SELECT salary, hire_date INTO sal, hiredate FROM employees
    WHERE employee_id = empid;
  IF hiredate > TO_DATE('01-JAN-00') THEN
     bonus := sal/20;
     DBMS_OUTPUT.PUT_LINE('Bonus for employee: ' || empid || ' is: ' || bonus );
  END IF;
END;
/
```

Example 4–17 shows the use of `IF-THEN-ELSEIF-ELSE` to determine the salary raise an employee receives based on the hire date of the employee.

Example 4–17 Using the IF-THEN-ELSEIF Statement in PL/SQL

```
DECLARE
  bonus    NUMBER(6,2);
  empid    NUMBER(6) := 120;
  hiredate DATE;
BEGIN
-- retrieve the date that employee was hired, the date is checked
-- to determine the amount of the bonus for the employee
  SELECT hire_date INTO hiredate FROM employees WHERE employee_id = empid;
  IF hiredate > TO_DATE('01-JAN-98') THEN
      bonus := 500;
   ELSIF hiredate > TO_DATE('01-JAN-96') THEN
      bonus := 1000;
   ELSE
      bonus := 1500;
   END IF;
   DBMS_OUTPUT.PUT_LINE('Bonus for employee: ' || empid || ' is: ' || bonus );
END;
/
```

Conditional Control With the CASE Statement

To choose among several values or courses of action, you can use `CASE` constructs. The `CASE` expression evaluates a condition and returns a value for each case. The case statement evaluates a condition, and performs an action, such as an entire PL/SQL block, for each case. When possible, rewrite lengthy `IF-THEN-ELSIF` statements as `CASE` statements because the `CASE` statement is more readable and more efficient.

Example 4–18 shows a simple `CASE` statement.

Example 4–18 Using the CASE-WHEN Statement in PL/SQL

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';
  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

Example 4–19 determines the salary raise an employee receives based on the current salary of the employee and the job ID. This complex example combines the CASE expression with IF-THEN-ELSE statements.

Example 4–19 Using the IF-THEN_ELSE and CASE Statement in PL/SQL

```
DECLARE -- declare variables
   empid          NUMBER(6) := 115;
   jobid          VARCHAR2(10);
   sal            NUMBER(8,2);
   sal_raise      NUMBER(3,2); -- this is the rate of increase for the raise
BEGIN
-- retrieve the job ID and salary for the employee and
-- assign the values to variables jobid and sal
  SELECT job_id, salary INTO jobid, sal from employees WHERE employee_id = empid;
  CASE  -- determine the salary raise rate based on employee job ID
    WHEN jobid = 'PU_CLERK' THEN
        IF sal < 3000 THEN sal_raise := .08;
          ELSE sal_raise := .07;
        END IF;
    WHEN jobid = 'SH_CLERK' THEN
        IF sal < 4000 THEN sal_raise := .06;
          ELSE sal_raise := .05;
        END IF;
    WHEN jobid = 'ST_CLERK' THEN
        IF sal < 3500 THEN sal_raise := .04;
          ELSE sal_raise := .03;
        END IF;
    ELSE
     BEGIN
-- if no conditions met, then the following
       DBMS_OUTPUT.PUT_LINE('No raise for this job: ' || jobid);
     END;
  END CASE;
-- display the percent raise for the employee
  DBMS_OUTPUT.PUT_LINE('Percent salary raise for employee: ' || empid || ' is: '
                       || sal_raise );
END;
/
```

A sequence of statements that uses query results to select alternative actions is common in database applications. Another common sequence inserts or deletes a row only if an associated entry is found in another table. You can bundle these common sequences into a PL/SQL block using conditional logic.

Iterative Control With LOOPs

LOOP statements let you run a sequence of statements multiple times. You place the keyword LOOP before the first statement in the sequence and the keywords END LOOP after the last statement in the sequence.

The FOR-LOOP statement lets you specify a range of integers, then run a sequence of statements once for each integer in the range. In Example 4–20, the loop displays the number and the square of the number for numbers 1 to 10. Note that you do not have to declare or initialize the counter in the FOR-LOOP and any valid identifier can be used for the name, such as loop_counter.

Example 4–20 Using the FOR-LOOP in PL/SQL

```
BEGIN
-- use a FOR loop to process a series of numbers
  FOR loop_counter IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE('Number: ' || TO_CHAR(loop_counter)
                         || ' Square: ' || TO_CHAR(loop_counter**2));
  END LOOP;
END;
/
```

The WHILE-LOOP statement associates a condition with a sequence of statements. Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement.

In Example 4–21, the loop displays the number and the cube of the number while the number is less than or equal to 10.

Example 4–21 Using WHILE-LOOP for Control in PL/SQL

```
DECLARE  -- declare variables
   i         NUMBER := 1; -- loop counter, initialize to one
   i_cubed   NUMBER;
BEGIN
-- use WHILE LOOP to process data
  WHILE i <= 10 LOOP
    i_cubed := i**3;
    DBMS_OUTPUT.PUT_LINE('Number: ' || TO_CHAR(i)
                      || ' Cube: ' || TO_CHAR(i_cubed));
    i := i + 1;
  END LOOP;
END;
/
```

The EXIT-WHEN statement lets you complete a loop if further processing is impossible or undesirable. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement. In Example 4–22, the loop completes when the value of total exceeds 25,000:

Example 4–22 Using the EXIT-WHEN Statement in PL/SQL

```
DECLARE -- declare and assign values to variables
  total   NUMBER(9) := 0;
  counter NUMBER(6) := 0;
BEGIN
  LOOP
    counter := counter + 1; -- increment counter variable
    total := total + counter * counter;  -- compute total
    -- exit loop when condition is true
    EXIT WHEN total > 25000; -- LOOP until condition is met
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Counter: ' || TO_CHAR(counter)
                       || ' Total: ' || TO_CHAR(total));  -- display results
END;
/
```

Sequential Control With GOTO

The GOTO statement lets you branch to a label unconditionally; however, you would usually try to avoid exiting a loop in this manner. The label, an undeclared identifier enclosed by double angle brackets, must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block.

Example 4–23 shows the use of the GOTO statement in a loop that is testing for prime numbers. When a number can be divided into evenly (no remainder), then it is not a prime and the loop is immediately exited. Note the use of the SQL numeric function MOD to check for no (zero) remainder. See "Using Numeric Functions" on page 3-14 for information about SQL numeric functions.

Example 4–23 Using the GOTO Statement in PL/SQL

```
DECLARE  -- declare variables
  p        VARCHAR2(30);
  n        PLS_INTEGER := 37; -- test any integer > 2 for prime, here 37
BEGIN
-- loop through divisors to determine if a prime number
  FOR j in 2..ROUND(SQRT(n))
  LOOP
    IF n MOD j = 0 THEN -- test for prime
      p := ' is NOT a prime number'; -- not a prime number
      GOTO print_now;
    END IF;
  END LOOP;
  p := ' is a prime number';
<<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);  -- display results
END;
/
```

Using Local PL/SQL Procedures and Functions in PL/SQL Blocks

Procedures and functions (subprograms) are named PL/SQL blocks that can be called with a set of parameters from inside of a PL/SQL block.

A procedure is a subprogram that performs a specific action. You specify the name of the procedure, its parameters, its local variables, and the BEGIN-END block that contains its code and handles any exceptions. A function is a subprogram that computes and returns a value. Functions and procedures are structured alike, except that functions return a value.

When passing parameters to functions and procedures, the parameters can be declared as IN or OUT or IN OUT parameters.

- IN indicates that you must supply a value for the argument when calling the function or procedure. This is the default.
- OUT indicates that the function or procedure will set the value of the argument.
- IN OUT indicates that a value for the argument can be supplied by you and can be set by the function or procedure.

Example 4–24 is an example of a declaration of a PL/SQL procedure in a PL/SQL block. Note that the v1 and v2 variables are declared as IN OUT parameters to a subprogram.

Example 4–24 Declaring a Local PL/SQL Procedure With IN OUT Parameters

```
DECLARE -- declare variables and subprograms
  fname    VARCHAR2(20) := 'randall';
  lname    VARCHAR2(25) := 'dexter';

-- declare a local procedure which can only be used in this block
  PROCEDURE upper_name ( v1 IN OUT VARCHAR2, v2 IN OUT VARCHAR2) AS
    BEGIN
      v1 := UPPER(v1); -- change the string to uppercase
      v2 := UPPER(v2); -- change the string to uppercase
    END upper_name;

-- start of executable part of block
BEGIN
  DBMS_OUTPUT.PUT_LINE(fname || ' ' || lname ); -- display initial values
  upper_name (fname, lname); -- call the procedure with parameters
  DBMS_OUTPUT.PUT_LINE(fname || ' ' || lname ); -- display new values
END;
/
```

Example 4–25 is an example of a declaration of a PL/SQL function in a PL/SQL block. Note that the value returned by the function is used directly in the DBMS_OUTPUT.PUT_LINE statement. Note that the v1 and v2 variables are declared as IN parameters to a subprogram. An IN parameter passes an initial value that is read inside of a subprogram. Any update to the value of the parameter inside of the subprogram is not accessible outside of the subprogram.

Example 4–25 Declaring a Local PL/SQL Function With IN Parameters

```
DECLARE -- declare variables and subprograms
  fname    VARCHAR2(20) := 'randall';
  lname    VARCHAR2(25) := 'dexter';

-- declare local function which can only be used in this block
  FUNCTION upper_name ( v1 IN VARCHAR2, v2 IN VARCHAR2)
    RETURN VARCHAR2 AS
    v3     VARCHAR2(45);  -- this variable is local to the function
    BEGIN
    -- build a string that will be returned as the function value
      v3 := v1 || ' + ' || v2 || ' = ' || UPPER(v1) || ' ' || UPPER(v2);
      RETURN v3;  -- return the value of v3
    END upper_name;

-- start of executable part of block
BEGIN
-- call the function and display results
  DBMS_OUTPUT.PUT_LINE(upper_name (fname, lname));
END;
/
```

In Example 4–26, both a variable and a numeric literal are passed as a parameter to a more complex procedure.

Example 4–26 Declaring a Complex Local Procedure in a PL/SQL Block

```
DECLARE  -- declare variables and subprograms
  empid NUMBER;

-- declare local procedure for this block
  PROCEDURE avg_min_max_sal (empid IN NUMBER) IS
    jobid     VARCHAR2(10);
    avg_sal   NUMBER;
    min_sal   NUMBER;
    max_sal   NUMBER;
  BEGIN
    -- determine the job ID for the employee
    SELECT job_id INTO jobid FROM employees WHERE employee_id = empid;
    -- calculate the average, minimum, and maximum salaries for that job ID
    SELECT AVG(salary), MIN(salary), MAX(salary) INTO avg_sal, min_sal, max_sal
      FROM employees WHERE job_id = jobid;
    -- display data
    DBMS_OUTPUT.PUT_LINE ('Employee ID: ' || empid || ' Job ID: ' || jobid);
    DBMS_OUTPUT.PUT_LINE ('The average salary for job ID: ' || jobid
                            || ' is ' || TO_CHAR(avg_sal));
    DBMS_OUTPUT.PUT_LINE ('The minimum salary for job ID: ' || jobid
                            || ' is ' || TO_CHAR(min_sal));
    DBMS_OUTPUT.PUT_LINE ('The maximum salary for job ID: ' || jobid
                            || ' is ' || TO_CHAR(max_sal));
  END avg_min_max_sal;
-- end of local procedure

-- start executable part of block
BEGIN
-- call the procedure with several employee IDs
  empid := 125;
  avg_min_max_sal(empid);
  avg_min_max_sal(112);
END;
/
```

Subprograms can also be declared in packages. For an example of a subprogram declaration in a package, see Example 5–9 on page 5-17. You can create standalone subprograms that are stored in the database. These subprograms can be called from other subprograms, packages, and SQL statements. See Chapter 5, "Using Procedures, Functions, and Packages".

Using Cursors and Cursor Variables To Retrieve Data

A cursor is a name for a private SQL area in which information for processing the specific statement is kept. PL/SQL uses both implicit and explicit cursors. Cursor attributes return useful information about the status of cursors in the execution of SQL statements.

PL/SQL implicitly creates a cursor for all SQL data manipulation statements on a set of rows, including queries that return only one row. Implicit cursors are managed automatically by PL/SQL so you are not required to write any code to handle these cursors. However, you can track information about the execution of an implicit cursor through its cursor attributes.

You can explicitly declare a cursor for one row or multiple rows if you want precise control over query processing. You must declare an explicit cursor for queries that return more than one row. For queries that return multiple rows, you can process the rows individually.

A cursor variable (REF CURSOR) is similar to a cursor and points to the current row in the result set of a multi-row query.

This section contains the following topics:

- Explicit Cursors on page 4-20
- Cursor Variables (REF CURSORs) on page 4-22
- Cursor Attributes on page 4-23

Explicit Cursors

Example 4–27 is an example of explicit cursor used to process one row of a table.You should explicitly open and close a cursor before and after use.

Example 4–27 Fetching a Single Row With a Cursor in PL/SQL

```
DECLARE
-- declare variables for first_name and last_name fetched from the employees table
  firstname  employees.first_name%TYPE;   -- variable for first_name
  lastname   employees.last_name%TYPE;   -- variable for last_name

-- declare a cursor to fetch data from a row (employee 120) in the employees table
  CURSOR cursor1 IS
    SELECT first_name, last_name FROM employees WHERE employee_id = 120;

BEGIN
  OPEN cursor1; -- open the cursor
  FETCH cursor1 INTO firstname, lastname; -- fetch data into local variables
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || firstname || ' ' || lastname);
  CLOSE cursor1; -- close the cursor
END;
/
```

Example 4–28 shows examples of the use of a cursor to process multiple rows in a table. The FETCH statement retrieves the rows in the result set one at a time. Each fetch retrieves the current row and advances the cursor to the next row in the result set. Note the use of the cursor attributes %ROWCOUNT and %NOTFOUND. For information about cursor attributes, see "Cursor Attributes" on page 4-23.

Example 4–28 Fetching Multiple Rows With a Cursor in PL/SQL

```
DECLARE
-- declare variables for data fetched from cursors
  empid      employees.employee_id%TYPE; -- variable for employee_id
  jobid      employees.job_id%TYPE;      -- variable for job_id
  lastname   employees.last_name%TYPE;   -- variable for last_name
  rowcount   NUMBER;
-- declare the cursors
  CURSOR cursor1 IS SELECT last_name, job_id FROM employees
                 WHERE job_id LIKE '%CLERK';
  CURSOR cursor2 is SELECT employee_id, last_name, job_id FROM employees
                 WHERE job_id LIKE '%MAN' OR job_id LIKE '%MGR';
BEGIN
-- start the processing with cursor1
  OPEN cursor1; -- open cursor1 before fetching
  DBMS_OUTPUT.PUT_LINE( '---------- cursor 1-----------------' );
  LOOP
    FETCH cursor1 INTO lastname, jobid; -- fetches 2 columns into variables
-- check the cursor attribute NOTFOUND for the end of data
```

```
    EXIT WHEN cursor1%NOTFOUND;
-- display the last name and job ID for each record (row) fetched
    DBMS_OUTPUT.PUT_LINE( RPAD(lastname, 25, ' ') || jobid );
  END LOOP;
  rowcount := cursor1%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE('The number of rows fetched is ' || rowcount );
  CLOSE cursor1;

-- start the processing with cursor2
  OPEN cursor2;
  DBMS_OUTPUT.PUT_LINE( '---------- cursor 2-----------------' );
  LOOP
-- fetch 3 columns into the variables
    FETCH cursor2 INTO empid, lastname, jobid;
    EXIT WHEN cursor2%NOTFOUND;
-- display the employee ID, last name, and job ID for each record (row) fetched
    DBMS_OUTPUT.PUT_LINE( empid || ': ' || RPAD(lastname, 25, ' ') || jobid );
  END LOOP;
  rowcount := cursor2%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE('The number of rows fetched is ' || rowcount );
  CLOSE cursor2;
END;
/
```

In Example 4–28, the `LIKE` condition operator is used to specify the records to return with the query. For information about `LIKE`, see "Restricting Data Using the WHERE Clause" on page 3-6.

Example 4–29 shows how to pass a parameter to an explicit cursor. In the example, the current month value is passed to the cursor to specify that only those employees hired during this month are displayed. This provides a list of employees that have their yearly anniversary dates and their bonus amount.

Example 4–29 Passing Parameters to a Cursor in PL/SQL

```
DECLARE
-- declare variables for data fetched from cursor
  empid       employees.employee_id%TYPE; -- variable for employee_id
  hiredate    employees.hire_date%TYPE;   -- variable for hire_date
  firstname   employees.first_name%TYPE;  -- variable for first_name
  lastname    employees.last_name%TYPE;   -- variable for last_name
  rowcount    NUMBER;
  bonusamount NUMBER;
  yearsworked NUMBER;
-- declare the cursor with a parameter,
  CURSOR cursor1 (thismonth NUMBER)IS
    SELECT employee_id, first_name, last_name, hire_date FROM employees
       WHERE EXTRACT(MONTH FROM hire_date) = thismonth;
BEGIN
-- open and pass a parameter to cursor1, select employees hired on this month
  OPEN cursor1(EXTRACT(MONTH FROM SYSDATE));
  DBMS_OUTPUT.PUT_LINE('----- Today is ' || TO_CHAR(SYSDATE, 'DL') || ' -----');
  DBMS_OUTPUT.PUT_LINE('Employees with yearly bonus amounts:');
  LOOP
-- fetches 4 columns into variables
    FETCH cursor1 INTO empid, firstname, lastname, hiredate;
-- check the cursor attribute NOTFOUND for the end of data
    EXIT WHEN cursor1%NOTFOUND;
-- calculate the yearly bonus amount based on months (years) worked
  yearsworked := ROUND( (MONTHS_BETWEEN(SYSDATE, hiredate)/12) );
  IF yearsworked > 10   THEN bonusamount := 2000;
  ELSIF yearsworked > 8 THEN bonusamount := 1600;
  ELSIF yearsworked > 6 THEN bonusamount := 1200;
  ELSIF yearsworked > 4 THEN bonusamount := 800;
  ELSIF yearsworked > 2 THEN bonusamount := 400;
  ELSIF yearsworked > 0 THEN bonusamount := 100;
  END IF;
-- display the employee Id, first name, last name, hire date, and bonus
-- for each record (row) fetched
    DBMS_OUTPUT.PUT_LINE( empid || ' ' || RPAD(firstname, 21, ' ') ||
      RPAD(lastname, 26, ' ') || hiredate || TO_CHAR(bonusamount, '$9,999'));
  END LOOP;
  rowcount := cursor1%ROWCOUNT;
  DBMS_OUTPUT.PUT_LINE('The number of rows fetched is ' || rowcount );
  CLOSE cursor1;
END;
/
```

See Also: *Oracle Database PL/SQL User's Guide and Reference* for information about managing cursors with PL/SQL

Cursor Variables (REF CURSORs)

Cursor variables (REF CURSORs) are like pointers to result sets. A cursor variable is more flexible than a cursor because it is not tied to a specific query. You can open a cursor variable for any query that returns the correct set of columns.

Cursor variables are used when you want to perform a query in one function or procedure, and process the results in a different subprogram, possibly in a different language. A cursor variable has the datatype REF CURSOR, and is often referred to informally as a REF CURSOR.

A REF CURSOR can be declared with a return type (strong type) or without a return type (weak type). A strong REF CURSOR type is less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with queries that return the right set of columns. A weak REF CURSOR types is more flexible because the compiler lets you associate a weakly typed cursor variable with any query. Because there is no type checking with a weak REF CURSOR, all such types are interchangeable. Instead of creating a new type, you can use the predefined type SYS_REFCURSOR.

Example 4–30 show how to declare a cursor variable of REF CURSOR datatype, then use that cursor variable as a formal parameter in a procedure. For additional examples of the use of REF CURSOR, see "Accessing Types in Packages" on page 5-21. For an example of the use of a REF CURSOR with a PHP program, see Appendix C, "Using a PL/SQL Procedure With PHP". For an example of the use of a REF CURSOR with a Java program, see Appendix D, "Using a PL/SQL Procedure With JDBC".

Example 4–30 Using a Cursor Variable (REF CURSOR)

```
DECLARE
-- declare a REF CURSOR that returns employees%ROWTYPE (strongly typed)
   TYPE emp_refcur_typ IS REF CURSOR RETURN employees%ROWTYPE;
   emp_cursor emp_refcur_typ;
-- use the following local procedure to process all the rows after
-- the result set is built, rather than calling a procedure for each row
   PROCEDURE process_emp_cv (emp_cv IN emp_refcur_typ) IS
      person employees%ROWTYPE;
   BEGIN
      DBMS_OUTPUT.PUT_LINE('-- Here are the names from the result set --');
      LOOP
         FETCH emp_cv INTO person;
         EXIT WHEN emp_cv%NOTFOUND;
         DBMS_OUTPUT.PUT_LINE(person.last_name || ', ' || person.first_name);
      END LOOP;
   END;
BEGIN
-- find employees whose employee ID is less than 108
  OPEN emp_cursor FOR SELECT * FROM employees WHERE employee_id < 108;
  process_emp_cv(emp_cursor); -- pass emp_cursor to the procedure for processing
  CLOSE emp_cursor;
-- find employees whose last name starts with R
  OPEN emp_cursor FOR SELECT * FROM employees WHERE last_name LIKE 'R%';
  process_emp_cv(emp_cursor);  -- pass emp_cursor to the procedure for processing
  CLOSE emp_cursor;
END;
/
```

See Also: *Oracle Database PL/SQL User's Guide and Reference* for information about using cursor variables (REF CURSORs)

Cursor Attributes

Cursor attributes return information about the execution of DML and DDL statements, such INSERT, UPDATE, DELETE, SELECT INTO, COMMIT, or ROLLBACK statements. The cursor attributes are %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. These attributes return useful information about the most recently executed SQL statement. When using an explicit cursor, add the explicit cursor or cursor variable name to the beginning of the attribute, such as cursor1%FOUND, to return information for the most recently executed SQL statement for that cursor.

The attributes provide the following information:

- %FOUND Attribute: Has a Row Been Fetched?

  After a cursor or cursor variable is opened but before the first fetch, %FOUND returns NULL. After any fetches, it returns TRUE if the last fetch returned a row, or FALSE if the last fetch did not return a row.

- %ISOPEN Attribute: Is the Cursor Open?

  If a cursor or cursor variable is open, then %ISOPEN returns TRUE ; otherwise, %ISOPEN returns FALSE.

  Note that implicit cursors are automatically opened before and closed after executing the associated SQL statement so %ISOPEN always returns FALSE.

- %NOTFOUND Attribute: Has a Fetch Failed?

  If the last fetch returned a row, then %NOTFOUND returns FALSE. If the last fetch failed to return a row, then %NOTFOUND returns TRUE. %NOTFOUND is the logical opposite of %FOUND.

- %ROWCOUNT Attribute: How Many Rows Fetched So Far?

  After a cursor or cursor variable is opened, %ROWCOUNT returns 0 before the first fetch. Thereafter, it returns the number of rows fetched so far. The number is incremented if the last fetch returned a row.

See Also: *Oracle Database PL/SQL User's Guide and Reference* for information about cursor attributes

Working With PL/SQL Data Structures

Data structure are composite datatypes that let you work with the essential properties of data without being too involved with details. After you design a data structure, you can focus on designing algorithms that manipulate the data structure.

This section contains the following topics:

- Using Record Types on page 4-24
- Using Collections on page 4-25

Using Record Types

Record types are composite data structures whose fields can have different datatypes. You can use records to hold related items and pass them to subprograms with a single parameter. When declaring records, you use the TYPE definition, as shown in Example 4–31.

Usually you would use a record to hold data from an entire row of a database table. You can use the %ROWTYPE attribute to declare a record that represents a row in a table or a row from a query result set, without specifying the names and types for the fields. When using %ROWTYPE, the record type definition is implied, and the TYPE keyword is not necessary, as shown in Example 4–32.

Example 4–31 shows how are records are declared and initialized.

Example 4–31 Declaring and Initializing a PL/SQL Record Type

```
DECLARE  -- declare RECORD type variables
-- the following is a RECORD declaration to hold address information
   TYPE location_rec IS RECORD (
        room_number     NUMBER(4),
        building        VARCHAR2(25)
        );
-- you use the %TYPE attribute to declare the datatype of a table column
-- you can include (nest) a record inside of another record
   TYPE person_rec IS RECORD (
        employee_id  employees.employee_id%TYPE,
        first_name   employees.first_name%TYPE,
        last_name    employees.last_name%TYPE,
        location     location_rec
        );
  person  person_rec; -- declare a person variable of type person_rec
BEGIN
-- insert data in a record, one field at a time
  person.employee_id := 20;
  person.first_name := 'James';
  person.last_name := 'Boynton';
  person.location.room_number := 100;
  person.location.building:= 'School of Education';
-- display data in a record
  DBMS_OUTPUT.PUT_LINE( person.last_name || ', ' || person.first_name );
  DBMS_OUTPUT.PUT_LINE( TO_CHAR(person.location.room_number) || ' '
                     || person.location.building );
END;
/
```

Example 4–32 shows the use of %ROWTYPE in a record type declaration. This record is used with a cursor that fetches an entire row.

Example 4–32 Using %ROWTYPE With a Cursor When Declaring a PL/SQL Record

```
DECLARE -- declare variables
  CURSOR cursor1 IS
    SELECT * FROM employees
      WHERE department_id = 60; -- declare cursor
-- declare record variable that represents a row fetched from the employees table
-- do not need to use TYPE .. IS RECORD with %ROWTYPE attribute
   employee_rec cursor1%ROWTYPE;
BEGIN
-- open the explicit cursor c1 and use it to fetch data into employee_rec
  OPEN cursor1;
  LOOP
    FETCH cursor1 INTO employee_rec; -- retrieve entire row into record
    EXIT WHEN cursor1%NOTFOUND;
-- the record contains all the fields for a row in the employees table
-- the following displays the data from the row fetched into the record
   DBMS_OUTPUT.PUT_LINE( ' Department ' || employee_rec.department_id
     || ', Employee: ' || employee_rec.employee_id || ' - '
     || employee_rec.last_name || ', ' || employee_rec.first_name );
  END LOOP;
  CLOSE cursor1;
END;
/
```

Example 4–34 on page 4-26 shows the use of record as an element in a varray.

> **See Also:** *Oracle Database PL/SQL User's Guide and Reference* for information about PL/SQL records

Using Collections

PL/SQL collection types let you declare high-level datatypes similar to arrays, sets, and hash tables found in other languages. In PL/SQL, array types are known as varrays (short for variable-size arrays), set types are known as nested tables, and hash table types are known as associative arrays. Each kind of collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. When declaring collections, you use a `TYPE` definition. To reference an element, use subscript notation with parentheses.

Example 4–33 shows the use of a varray with elements of character type. A varray must be initialized before use. When initializing a varry, you can also insert values into the elements. After initialization, you need to use `EXTEND` to add additional elements before inserting more values into the varray.

Example 4–33 Using a PL/SQL VARRAY Type With Character Elements

```
DECLARE -- declare variables
  TYPE jobids_array IS VARRAY(20) OF VARCHAR2(10);  -- declare VARRAY
  jobids  jobids_array; -- declare a variable of type jobids_array
  howmany NUMBER;  -- declare a variable to hold employee count
BEGIN
  -- initialize the arrary with some job ID values
  jobids := jobids_array('AC_ACCOUNT', 'AC_MGR', 'AD_ASST', 'AD_PRES', 'AD_VP',
                       'FI_ACCOUNT', 'FI_MGR', 'HR_REP', 'IT_PROG', 'PU_MAN',
                       'SH_CLERK', 'ST_CLERK', 'ST_MAN');
-- display the current size of the array with COUNT
  DBMS_OUTPUT.PUT_LINE('The number of elements (current size) in the array is '
                      || jobids.COUNT);
-- display the maximum number of elements for the array LIMIT
  DBMS_OUTPUT.PUT_LINE('The maximum number (limit) of elements in the array is '
                        || jobids.LIMIT);
-- check whether another element can be added to the array
  IF jobids.LIMIT - jobids.COUNT >= 1 THEN
     jobids.EXTEND(1); -- add one more element
     jobids(14) := 'PU_CLERK';  -- assign a value to the element
  END IF;
-- loop through all the varray values, starting
-- with the FIRST and ending with the LAST element
  FOR i IN jobids.FIRST..jobids.LAST LOOP
  -- determine the number of employees for each job ID in the array
    SELECT COUNT(*) INTO howmany FROM employees WHERE job_id = jobids(i);
    DBMS_OUTPUT.PUT_LINE ( 'Job ID: ' || RPAD(jobids(i), 10, ' ') ||
                           ' Number of employees: ' || TO_CHAR(howmany));
  END LOOP;
-- display the current size of the array with COUNT
  DBMS_OUTPUT.PUT_LINE('The number of elements (current size) in the array is '
                        || jobids.COUNT);
END;
/
```

Example 4–34 shows the use of a varray with record type elements.

Example 4–34 Using a PL/SQL VARRAY Type With Record Type Elements

```
DECLARE -- declare variables
  CURSOR cursor1 IS SELECT * FROM jobs; -- create a cursor for fetching the rows
  jobs_rec  cursor1%ROWTYPE; -- create a record to hold the row data
 -- declare VARRAY with enough elements to hold all the rows in the jobs table
  TYPE jobs_array IS VARRAY(25) OF cursor1%ROWTYPE;
  jobs_arr  jobs_array; -- declare a variable of type jobids_array
  howmany   NUMBER;  -- declare a variable to hold employee count
  i         NUMBER := 1; -- counter for the number of elements in the array
BEGIN
  jobs_arr := jobs_array(); -- initialize the array before using
  OPEN cursor1; -- open the cursor before using
  LOOP
    FETCH cursor1 INTO jobs_rec; -- retrieve a row from the jobs table
    EXIT WHEN cursor1%NOTFOUND; -- exit when no data is retrieved
    jobs_arr.EXTEND(1); -- add another element to the varray with EXTEND
    jobs_arr(i) := jobs_rec; -- assign the fetched row to an element the array
    i := i + 1; -- increment the element count
  END LOOP;
  CLOSE cursor1; -- close the cursor when finished with it
  FOR j IN jobs_arr.FIRST..jobs_arr.LAST LOOP -- loop through the varray elements
  -- determine the number of employees for each job ID in the array
    SELECT COUNT(*) INTO howmany FROM employees WHERE job_id = jobs_arr(j).job_id;
    DBMS_OUTPUT.PUT_LINE ( 'Job ID: ' || RPAD(jobs_arr(j).job_id, 11, ' ') ||
                           RPAD(jobs_arr(j).job_title, 36, ' ') ||
                           ' Number of employees: ' || TO_CHAR(howmany));
  END LOOP;
END;
/
```

Collections can be passed as parameters, so that subprograms can process arbitrary numbers of elements.

See Also: *Oracle Database PL/SQL User's Guide and Reference* for information about PL/SQL collections

Using Bind Variables With PL/SQL

When you embed an INSERT, UPDATE, DELETE, or SELECT SQL statement directly in your PL/SQL code, PL/SQL turns the variables in the WHERE and VALUES clauses into bind variables automatically. Oracle Database XE can reuse these SQL statement each time the same code is executed. When running similar statements with different variable values, you can improve performance by calling a stored procedure that accepts parameters, then issues the statements with the parameters substituted in the appropriate places.

You need to specify bind variables with dynamic SQL, in clauses such as WHERE and VALUES where you normally use variables. Instead of concatenating literals and variable values into a single string, replace the variables with the names of bind variables (preceded by a colon), and specify the corresponding PL/SQL variables with the USING clause. Using the USING clause, instead of concatenating the variables into the string, reduces parsing overhead and lets Oracle Database XE reuse the SQL statements.

In Example 4–35, :dptid, :dptname, :mgrid, and :locid are examples of bind variables.

See Also: "About Bind Variables" in *Oracle Database Express Edition Application Express User's Guide*

Using Dynamic SQL in PL/SQL

PL/SQL supports both dynamic and static SQL. Dynamic SQL enables you to build SQL statements dynamically at run time while static SQL statements are known in advance. You can create more general-purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation time.

To process most dynamic SQL statements, you use the EXECUTE IMMEDIATE statement. Dynamic SQL is especially useful for executing SQL statements to create database objects, such as CREATE TABLE.

Example 4–35 shows an example of the use of dynamic SQL to manipulate data in a table.

Example 4–35 Using Dynamic SQL to Manipulate Data in PL/SQL

```
DECLARE
   sql_stmt          VARCHAR2(200); -- variable to hold SQL statement
   column_name       VARCHAR2(30);  -- variable for column name
   dept_id           NUMBER(4);
   dept_name         VARCHAR2(30);
   mgr_id            NUMBER(6);
   loc_id            NUMBER(4);
BEGIN
-- create a SQL statement (sql_stmt) to execute with EXECUTE IMMEDIATE
-- the statement INSERTs a row into the departments table using bind variables
-- note that there is no semi-colon (;) inside the quotation marks '...'
  sql_stmt := 'INSERT INTO departments VALUES (:dptid, :dptname, :mgrid, :locid)';
  dept_id := 46;
  dept_name := 'Special Projects';
  mgr_id := 200;
  loc_id := 1700;
-- execute the sql_stmt using the values of the variables in the USING clause
-- for the bind variables
  EXECUTE IMMEDIATE sql_stmt USING dept_id, dept_name, mgr_id, loc_id;
```

```
-- use EXECUTE IMMEDIATE to delete the row that was previously inserted,
-- substituting for the column name and using a bind variable
  column_name := 'DEPARTMENT_ID';
  EXECUTE IMMEDIATE 'DELETE FROM departments WHERE ' || column_name  || ' = :num'
      USING dept_id;
END;
/
```

Example 4–36 is an example of the use of dynamic SQL to create a table. For a more complete example, see Example 5–3 on page 5-9.

Example 4–36 Using Dynamic SQL to Create a Table in PL/SQL

```
DECLARE
  tabname       VARCHAR2(30); -- variable for table name
  current_date  VARCHAR2(8);  -- varible for current date
BEGIN
-- extract, format, and insert the year, month, and day from SYSDATE into
-- the current_date variable
  SELECT TO_CHAR(EXTRACT(YEAR FROM SYSDATE)) ||
     TO_CHAR(EXTRACT(MONTH FROM SYSDATE),'FM09') ||
     TO_CHAR(EXTRACT(DAY FROM SYSDATE),'FM09') INTO current_date FROM DUAL;
-- construct the table name with the current date as a suffix
  tabname := 'log_table_' || current_date;
-- use EXECUTE IMMEDIATE to create a table with tabname as the table name
  EXECUTE IMMEDIATE 'CREATE TABLE ' || tabname ||
                    '(op_time VARCHAR2(10), operation VARCHAR2(50))' ;
  DBMS_OUTPUT.PUT_LINE(tabname || ' has been created');
-- now drop the table
  EXECUTE IMMEDIATE 'DROP TABLE ' || tabname;
END;
/
```

See Also: *Oracle Database Application Developer's Guide - Fundamentals* for additional information about dynamic SQL

Handling PL/SQL Errors

PL/SQL makes it easy to detect and process error conditions known as exceptions. When an error occurs, an exception is raised: normal processing stops, and control transfers to special exception-handling code, which comes at the end of any PL/SQL block. Each different exception is processed by a particular exception handler.

The exception handling for PL/SQL is different from the manual checking you might be used to from C programming, where you insert a check to make sure that every operation succeeded. Instead, the checks and calls to error routines are performed automatically, similar to the exception mechanism in Java programming.

Predefined exceptions are raised automatically for certain common error conditions involving variables or database operations. For example, if you try to divide a number by zero, PL/SQL raises the predefined exception ZERO_DIVIDE automatically. See "Summary of Predefined PL/SQL Exceptions" on page 4-29.

You can declare exceptions of your own, for conditions that you decide are errors, or to correspond to database errors that normally result in ORA- error messages. When you detect a user-defined error condition, you execute a RAISE statement. See "Declaring PL/SQL Exceptions" on page 4-30.

This section contains the following topics:

- Summary of Predefined PL/SQL Exceptions on page 4-29
- Using the Exception Handler on page 4-30
- Declaring PL/SQL Exceptions on page 4-30
- Scope Rules for PL/SQL Exceptions on page 4-31
- Continuing After an Exception Is Raised on page 4-31

See Also: *Oracle Database PL/SQL User's Guide and Reference* for information about handling PL/SQL errors

Summary of Predefined PL/SQL Exceptions

An internal exception is raised automatically if your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. In PL/SQL common Oracle errors are predefined as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows. To handle unexpected Oracle errors, you can use the OTHERS handler.

PL/SQL declares predefined exceptions globally in package STANDARD so you do not need to declare them. You can write handlers for predefined exceptions using the predefined names. Table 4–1 lists some of the predefined exceptions.

Table 4–1 Predefined PL/SQL Exceptions

Exception	Description
ACCESS_INTO_NULL	A program attempts to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	A program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	A program attempts to open a cursor that is already open. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	A program attempts to store duplicate values in a column that is constrained by a unique index.
INVALID_CURSOR	A program attempts a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.) This exception is also raised when the LIMIT-clause expression in a bulk FETCH statement does not evaluate to a positive number.
LOGIN_DENIED	A program attempts to log on to Oracle Database XE with a user name or password that is not valid.
NO_DATA_FOUND	A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table. Because this exception is used internally by some SQL functions to signal completion, do not rely on this exception being propagated if you raise it within a function that is called as part of a query.
NOT_LOGGED_ON	A program issues a database call without being connected to Oracle Database XE.

Table 4–1 (Cont.) Predefined PL/SQL Exceptions

Exception	Description
`ROWTYPE_MISMATCH`	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. When an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
`SUBSCRIPT_BEYOND_COUNT`	A program references a nested table or varray element using an index number larger than the number of elements in the collection.
`SUBSCRIPT_OUTSIDE_LIMIT`	A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
`TOO_MANY_ROWS`	A `SELECT INTO` statement returns more than one row.
`VALUE_ERROR`	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL cancels the assignment and raises `VALUE_ERROR`. In procedural statements, `VALUE_ERROR` is raised if the conversion of a character string into a number fails. (In SQL statements, `INVALID_NUMBER` is raised.)
`ZERO_DIVIDE`	A program attempts to divide a number by zero.

Using the Exception Handler

Using exceptions for error handling has several advantages. With exceptions, you can reliably handle potential errors from many statements with a single exception handler, as shown in Example 4–37.

Example 4–37 Managing Multiple Errors With a Single PL/SQL Exception Handler

```
DECLARE  -- declare variables
   emp_column       VARCHAR2(30) := 'last_name';
   table_name       VARCHAR2(30) := 'emp';  -- set value to raise error
   temp_var         VARCHAR2(30);
BEGIN
  temp_var := emp_column;
  SELECT COLUMN_NAME INTO temp_var FROM USER_TAB_COLS
    WHERE TABLE_NAME = 'EMPLOYEES' AND COLUMN_NAME = UPPER(emp_column);
-- processing here
  temp_var := table_name;
  SELECT OBJECT_NAME INTO temp_var FROM USER_OBJECTS
    WHERE OBJECT_NAME = UPPER(table_name) AND OBJECT_TYPE = 'TABLE';
-- processing here
EXCEPTION
   WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
     DBMS_OUTPUT.PUT_LINE ('No Data found for SELECT on ' || temp_var);
END;
/
```

Declaring PL/SQL Exceptions

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the `EXCEPTION` keyword. In Example 4–38, you declare an exception named `past_due` that is raised when the `due_date` is less than the today's date.

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

Scope Rules for PL/SQL Exceptions

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its subblocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a subblock.

If you redeclare a global exception in a subblock, the local declaration prevails. The subblock cannot reference the global exception, unless the exception is declared in a labeled block and you qualify its name with the block label, for example:

`block_label.exception_name`

Example 4–38 shows the scope rules.

Example 4–38 Determining the Scope of PL/SQL Exceptions

```
DECLARE
   past_due EXCEPTION;
   acct_num NUMBER;
BEGIN
   DECLARE  ---------- subblock begins
     past_due EXCEPTION;  -- this declaration prevails
     acct_num NUMBER;
     due_date DATE := SYSDATE - 1; -- set on purpose to raise exception
     todays_date DATE := SYSDATE;
   BEGIN
      IF due_date < todays_date THEN
         RAISE past_due;  -- this is not handled
      END IF;
   END;  ------------- subblock ends
EXCEPTION
  WHEN past_due THEN  -- does not handle raised exception
    DBMS_OUTPUT.PUT_LINE('Handling PAST_DUE exception.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Could not recognize PAST_DUE_EXCEPTION in this scope.');
END;
/
```

The enclosing block does not handle the raised exception because the declaration of `past_due` in the subblock prevails. Although they share the same name, the two `past_due` exceptions are different, just as the two `acct_num` variables share the same name but are different variables. Thus, the `RAISE` statement and the `WHEN` clause refer to different exceptions. To have the enclosing block handle the raised exception, you must remove its declaration from the subblock or define an `OTHERS` handler.

Continuing After an Exception Is Raised

By default, you put an exception handler at the end of a subprogram to handle exceptions that are raised anywhere inside the subprogram. To continue execution from the spot where an exception occurred, enclose the code that might raise an exception inside another `BEGIN-END` block with its own exception handler. For example, put separate `BEGIN-END` blocks around groups of SQL statements that might raise `NO_DATA_FOUND`, or around arithmetic operations that might raise

DIVIDE_BY_ZERO. By putting a BEGIN-END block with an exception handler inside of a loop, you can continue executing the loop if some loop iterations raise exceptions.

You can still handle an exception for a statement, then continue with the next statement. Place the statement in its own subblock with its own exception handlers. If an error occurs in the subblock, a local handler can catch the exception. When the subblock ends, the enclosing block continues to execute at the point where the subblock ends, as shown in Example 4–39.

Example 4–39 Continuing After an Exception in PL/SQL

```
-- create a temporary table for this example
CREATE TABLE employees_temp AS
  SELECT employee_id, salary, commission_pct FROM employees;

DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp VALUES (303, 2500, 0);
  BEGIN -- subblock begins
    SELECT salary / commission_pct INTO sal_calc FROM employees_temp
      WHERE employee_id = 303;
    EXCEPTION
      WHEN ZERO_DIVIDE THEN
        sal_calc := 2500;
  END; -- subblock ends
  INSERT INTO employees_temp VALUES (304, sal_calc/100, .1);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    NULL;
END;
/
-- view the results
SELECT * FROM employees_temp WHERE employee_id = 303 OR employee_id = 304;
-- drop the temporary table
DROP TABLE employees_temp;
```

In this example, if the SELECT INTO statement raises a ZERO_DIVIDE exception, the local handler catches it and sets sal_calc to 2500. Execution of the handler is complete, so the subblock terminates, and execution continues with the INSERT statement.

5 Using Procedures, Functions, and Packages

This section discusses the development of procedures, functions, and packages with PL/SQL code, which was described in Chapter 4, "Using PL/SQL".

This section contains the following topics:

- Overview of Procedures, Functions, and Packages on page 5-1
- Managing Stored Procedures and Functions on page 5-3
- Managing Packages on page 5-13
- Oracle Provided Packages on page 5-23

See Also:

- "Using PL/SQL Packages" in *Oracle Database PL/SQL User's Guide and Reference* for additional information about PL/SQL packages
- "Using PL/SQL Subprograms" in *Oracle Database PL/SQL User's Guide and Reference* for information about PL/SQL subprograms

Overview of Procedures, Functions, and Packages

Oracle Database XE offers the capability to store programs in the database. This functionality enables commonly required code to be written and tested once and then accessed by any application that requires the code. Database-resident program units also ensure that the same processing is applied to the data when the code is invoked, making the development of applications easier and providing consistency between developers.

You can write database-resident programs in PL/SQL, and can use Object Browser to manage source types such as procedures, functions, and packages. The actions include creating, compiling, creating synonyms for, granting privileges on, and showing dependencies for these source types.

This chapter describes the main types of program units you can create with PL/SQL: procedures, functions, and packages. Procedures, functions, and packages are saved and stored in the database, and can be used as building blocks for applications.

For information about the features of the PL/SQL language, see Chapter 4, "Using PL/SQL".

This section contains the following topics:

- Stored Procedures and Functions on page 5-2
- Packages on page 5-2

See Also: *Oracle Database PL/SQL User's Guide and Reference* to learn about PL/SQL code and program units

Stored Procedures and Functions

Stored procedures and functions (subprograms) can be compiled and stored in an Oracle Database XE, ready to be executed. Once compiled, it is a schema object known as a stored procedure or stored function, which can be referenced or called any number of times by multiple applications connected to Oracle Database XE. Both stored procedures and functions can accept parameters when they are executed (called). To execute a stored procedure or function, you only need to include its object name.

Procedures and functions that are created outside of a package are called stored or standalone subprograms. Procedures and functions defined within a package are known as packaged subprograms. Procedures and functions nested inside other subprograms or within a PL/SQL block are known as local subprograms, which cannot be referenced by other applications and exist only inside of the enclosing block. For information about subprograms in PL/SQL blocks, see "Using Local PL/SQL Procedures and Functions in PL/SQL Blocks" on page 4-17.

Stored procedures and functions are the key to modular, reusable PL/SQL code. Wherever you might use a JAR file in Java, a module in Perl, a shared library in C++, or a DLL in Visual Basic, you can use PL/SQL stored procedures, stored functions, and packages.

You can call stored procedures or functions from a database trigger, another stored subprogram, or interactively from SQL Command Line (SQL*Plus). You can also configure a Web server so that the HTML for a Web page is generated by a stored subprogram, making it simple to provide a Web interface for data entry and report generation.

Procedures and functions are stored in a compact compiled form. When called, they are loaded and processed immediately. Subprograms take advantage of shared memory, so that only one copy of a subprogram is loaded into memory for execution by multiple users.

See Also: *Oracle Database Express Edition 2 Day DBA* for information about managing memory with Oracle Database XE

Packages

A package is a schema object that groups logically related PL/SQL types, variables, and subprograms. Packages usually have two parts, a specification (called the spec) and a body; sometimes the body is unnecessary. The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside of the package. The body defines the queries for the cursors and the code for the subprograms.

You can think of the specification as an interface and the body as a black box. You can debug, enhance, or replace a package body without changing the package specification.

The specification holds public declarations, which are visible to stored procedures and other code outside of the package. You must declare subprograms at the end of the specification.

The package body holds implementation details and private declarations, which are hidden from code outside of the package. Following the declarative part of the package body is the optional initialization part, which holds statements that initialize package variables and do any other one-time setup steps.

Applications that call the subprograms in a package only need to know the names and parameters from the package specification. You can change the implementation details inside the package body without affecting the calling applications.

Managing Stored Procedures and Functions

You can create, modify, run, and drop stored procedures and functions with the SQL Commands page, the Object Browser page, the Script Editor page, or SQL Command Line (SQL*Plus). You can view existing functions and procedures in Object Browser.

The SQL `CREATE PROCEDURE` statement is used to create stored procedures that are stored in the database. The SQL `CREATE FUNCTION` statement is used to create stored functions that are stored in an Oracle database.

A procedure or function is similar to a miniature program. It has an optional declarative part, an executable part, and an optional exception-handling part. A procedure is a subprogram that performs a specific action. You specify the name of the procedure, its parameters, its local variables, and the `BEGIN-END` block that contains its code and handles any exceptions. A function is a subprogram that computes and returns a value. Functions and procedures are structured alike, except that functions return a value. See "Using the PL/SQL Block Structure" on page 4-4.

When passing parameters to functions and procedures, the parameters can be declared as `IN` or `OUT` or `IN OUT` parameters. For a description of these parameter declarations, see "Using Local PL/SQL Procedures and Functions in PL/SQL Blocks" on page 4-17.

See Also:

- See "CREATE PROCEDURE" in *Oracle Database SQL Reference*
- See "CREATE FUNCTION" in *Oracle Database SQL Reference*

This section contains the following topics:

- Creating a Procedure or Function With the SQL Commands Page on page 5-4
- Creating a Procedure or Function With the Object Browser Page on page 5-5
- Viewing Procedures or Functions With the Object Browser Page on page 5-6
- Creating Stored Procedures With SQL CREATE PROCEDURE on page 5-7
- Creating a Stored Procedure That Uses Parameters on page 5-7
- Creating a Stored Procedure With the AUTHID Clause on page 5-9
- Creating Stored Functions With the SQL CREATE FUNCTION Statement on page 5-10
- Calling Stored Procedures or Functions on page 5-11
- Editing Procedures or Functions on page 5-12
- Dropping a Procedure or Function on page 5-13

See Also:

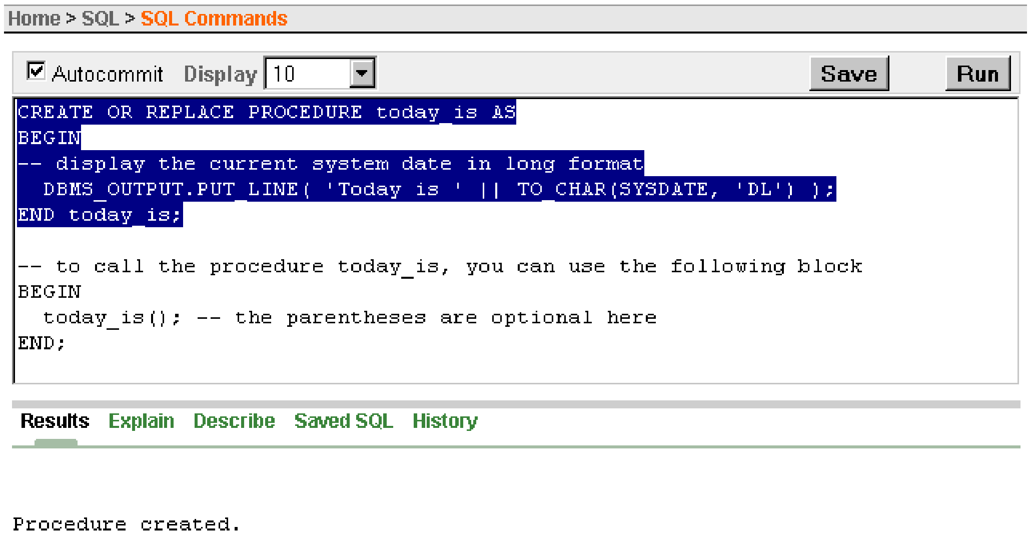
- *Oracle Database Express Edition Application Express User's Guide* for information about managing functions with Object Browser
- *Oracle Database Express Edition Application Express User's Guide* for information about managing procedures with Object Browser

Creating a Procedure or Function With the SQL Commands Page

You can use the SQL Commands page to create stored procedures or functions.

To create a procedure or function with the SQL Commands page:

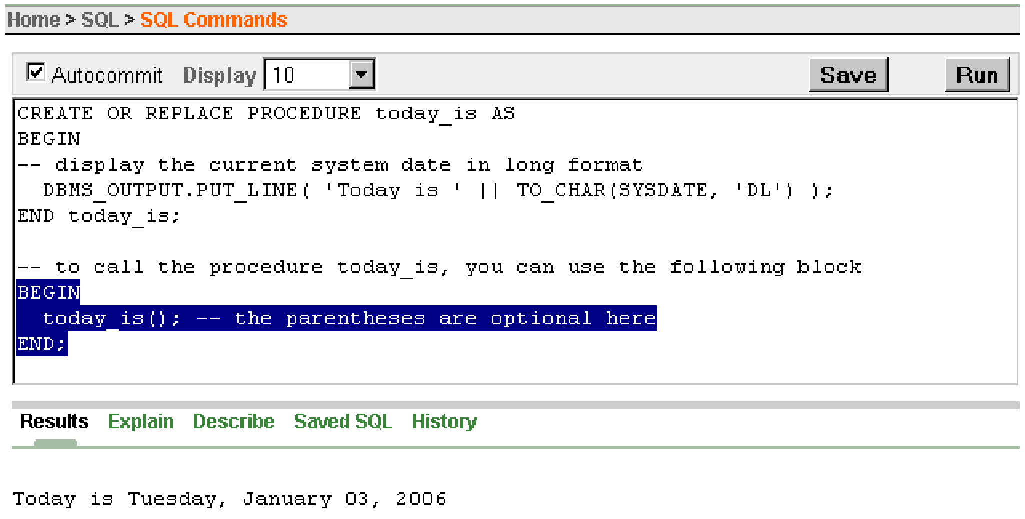
1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the home page, click the **SQL** icon to display the SQL page.
3. Click the **SQL Commands** icon to display the SQL Commands page.
4. On the SQL Commands page, enter the PL/SQL code for the PL/SQL procedure or function. You can use the code in Example 5–1 on page 5-7.
5. Select (highlight) the code for creating the procedure or function, then click the **Run** button to create the procedure or function.


Home > SQL > SQL Commands
Autocommit Display 10
Save
Run
Results Explain Describe Saved SQL History

```
CREATE OR REPLACE PROCEDURE today_is AS
BEGIN
-- display the current system date in long format
  DBMS_OUTPUT.PUT_LINE( 'Today is ' || TO_CHAR(SYSDATE, 'DL') );
END today_is;

-- to call the procedure today_is, you can use the following block
BEGIN
  today_is(); -- the parentheses are optional here
END;
```

```
Procedure created.
```

6. Select (highlight) the code for calling the procedure or function, then click the **Run** button to call the procedure or function.


Home > SQL > SQL Commands
Autocommit Display 10
Save
Run
Results Explain Describe Saved SQL History

```
CREATE OR REPLACE PROCEDURE today_is AS
BEGIN
-- display the current system date in long format
  DBMS_OUTPUT.PUT_LINE( 'Today is ' || TO_CHAR(SYSDATE, 'DL') );
END today_is;

-- to call the procedure today_is, you can use the following block
BEGIN
  today_is(); -- the parentheses are optional here
END;
```

```
Today is Tuesday, January 03, 2006
```

7. If you want to save the PL/SQL code for future use, click the **Save** button.
8. In the **Name** field, enter a name for the saved PL/SQL code. You can also enter an optional description. Click the **Save** button to save the code.
9. To access saved PL/SQL code, click the **Saved SQL** tab and select the name of the saved PL/SQL code that you want to access.

In the previous steps you created a procedure. For information about how to execute or call a procedure, see "Calling Stored Procedures or Functions" on page 5-11.

Creating a Procedure or Function With the Object Browser Page

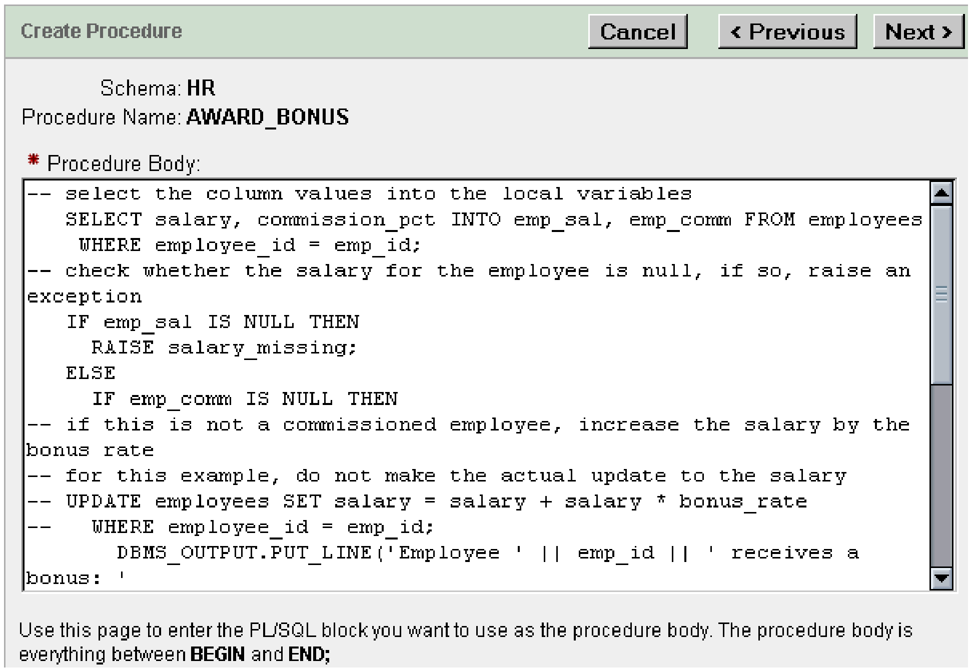
You can use the Object Browser page to create stored procedures or functions. This section explains how to create a procedure.

To create a procedure:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user `HR` with your password for the `HR` account.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the object list under **Create**, select **Procedure**.
4. Enter the procedure name (`award_bonus`), check the **Include Arguments** box, and then click the **Next** button.
5. Enter information for the arguments, then click **Next**. Use the arguments in Example 5–2 on page 5-7. For example:

```
Name   IN/OUT Type
emp_id      IN     NUMBER
bonus_rate  IN     NUMBER
```

6. Enter the source code for the procedure body, then click the **Next** button. Enter the PL/SQL source code in Example 5–2 on page 5-7.



Schema: **HR**
Procedure Name: **AWARD_BONUS**

* Procedure Body:

```
-- select the column values into the local variables
   SELECT salary, commission_pct INTO emp_sal, emp_comm FROM employees
    WHERE employee_id = emp_id;
-- check whether the salary for the employee is null, if so, raise an
exception
   IF emp_sal IS NULL THEN
     RAISE salary_missing;
   ELSE
     IF emp_comm IS NULL THEN
-- if this is not a commissioned employee, increase the salary by the
bonus rate
-- for this example, do not make the actual update to the salary
-- UPDATE employees SET salary = salary + salary * bonus_rate
--   WHERE employee_id = emp_id;
       DBMS_OUTPUT.PUT_LINE('Employee ' || emp_id || ' receives a
bonus: '
```

Use this page to enter the PL/SQL block you want to use as the procedure body. The procedure body is everything between **BEGIN** and **END;**

7. Click the **SQL** tab to view the source code for the procedure body. If you need to make corrections, click the **Previous** button.
8. When you have finished, click the **Finish** button. You can click the **Edit** button to make updates to the subprogram, such as adding additional variable declarations outside the `BEGIN` .. `END` block as in Example 5–2 on page 5-7.
9. Click the **Compile** button to compile the procedure. If errors occur, correct the source code and try compiling again. Compiling the procedure also saves any changes to the procedure.
10. When you have finished, click the **Finish** button.

In the previous steps, you created a procedure. For information about how to execute or call a procedure, see "Calling Stored Procedures or Functions" on page 5-11.

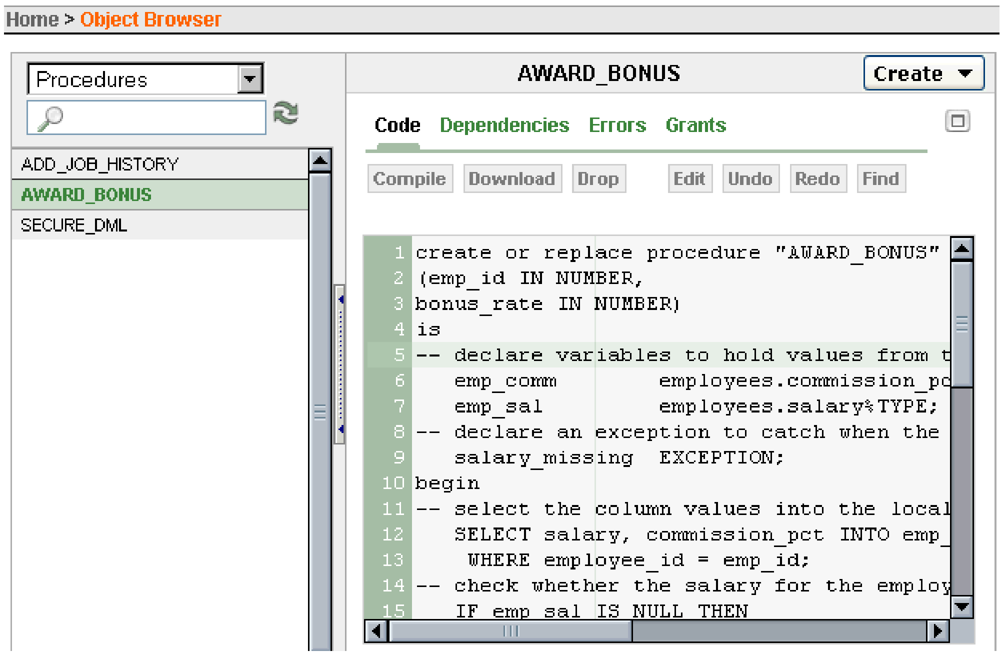
Viewing Procedures or Functions With the Object Browser Page

To find out which stored procedures or functions exist in your database, use the Object Browser.

To use Object Browser to view procedures and functions:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user `HR` with your password for the `HR` account.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the object list, select **Procedures** or **Functions**, then click the name of the procedure or function you want to display. For example, you could select **Procedures**, then click the name of the procedure (`AWARD_BONUS`) you previously created.

   The procedure or function information displays.



```
create or replace procedure "AWARD_BONUS"
(emp_id IN NUMBER,
bonus_rate IN NUMBER)
is
-- declare variables to hold values from t
   emp_comm        employees.commission_pc
   emp_sal         employees.salary%TYPE;
-- declare an exception to catch when the
   salary_missing  EXCEPTION;
begin
-- select the column values into the local
   SELECT salary, commission_pct INTO emp_
    WHERE employee_id = emp_id;
-- check whether the salary for the employ
   IF emp_sal IS NULL THEN
```

Creating Stored Procedures With SQL CREATE PROCEDURE

The SQL `CREATE PROCEDURE` statement lets you create stored procedures that are stored in the database. These stored (schema level) subprograms can be accessed from SQL. You can use the optional `OR REPLACE` clause to modify an existing procedure without first dropping the procedure.

Example 5–1 is an example of a simple stored procedure that displays current date.

Example 5–1 Creating a Simple Stored Procedure

```
CREATE OR REPLACE PROCEDURE today_is AS
BEGIN
-- display the current system date in long format
  DBMS_OUTPUT.PUT_LINE( 'Today is ' || TO_CHAR(SYSDATE, 'DL') );
END today_is;
/
-- to call the procedure today_is, you can use the following block
BEGIN
  today_is(); -- the parentheses are optional here
END;
/
```

Creating a Stored Procedure That Uses Parameters

When you create a procedure or function, you can specify parameters that are passed to the procedure or function when it is called (or invoked). In Example 5–2, note the use of the `IN` option with procedure arguments `emp_id` and `bonus_rate`. For a discussion of `IN` and `IN OUT` argument options in PL/SQL subprograms, see "Using Local PL/SQL Procedures and Functions in PL/SQL Blocks" on page 4-17.

Example 5–2 Creating a Stored Procedure That Uses Parameters

```
-- including OR REPLACE is more convenient when updating a subprogram
-- IN is the default for parameter declarations so it could be omitted
```

```
CREATE OR REPLACE PROCEDURE award_bonus (emp_id IN NUMBER, bonus_rate IN NUMBER)
  AS
-- declare variables to hold values from table columns, use %TYPE attribute
   emp_comm        employees.commission_pct%TYPE;
   emp_sal         employees.salary%TYPE;
-- declare an exception to catch when the salary is NULL
   salary_missing  EXCEPTION;
BEGIN  -- executable part starts here
-- select the column values into the local variables
   SELECT salary, commission_pct INTO emp_sal, emp_comm FROM employees
    WHERE employee_id = emp_id;
-- check whether the salary for the employee is null, if so, raise an exception
   IF emp_sal IS NULL THEN
     RAISE salary_missing;
   ELSE
     IF emp_comm IS NULL THEN
-- if this is not a commissioned employee, increase the salary by the bonus rate
-- for this example, do not make the actual update to the salary
-- UPDATE employees SET salary = salary + salary * bonus_rate
--   WHERE employee_id = emp_id;
       DBMS_OUTPUT.PUT_LINE('Employee ' || emp_id || ' receives a bonus: '
                           || TO_CHAR(emp_sal * bonus_rate) );
     ELSE
       DBMS_OUTPUT.PUT_LINE('Employee ' || emp_id
                           || ' receives a commission. No bonus allowed.');
     END IF;
   END IF;
EXCEPTION  -- exception-handling part starts here
   WHEN salary_missing THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' || emp_id ||
                          ' does not have a value for salary. No update.');
   WHEN OTHERS THEN
      NULL; -- for other exceptions do nothing
END award_bonus;
/

-- the following BEGIN..END block calls, or executes, the award_bonus procedure
-- using employee IDs 123 and 179 with the bonus rate 0.05 (5%)
BEGIN
  award_bonus(123, 0.05);
  award_bonus(179, 0.05);
END;
/
```

The output of the calls is similar to:

```
Employee 123 received a bonus: 325

Employee 179 receives a commission. No bonus allowed.
```

When executed, this procedure processes an employee ID and a bonus rate. It uses the Id to select the salary and commission percentage of the employee from the `employees` table. If the salary is null, an exception is raised. If the employee does not receive a commission, the employee's salary is updated by the bonus rate; otherwise no update is made. For a discussion of exception handling, see "Handling PL/SQL Errors" on page 4-28.

For different methods to execute (call) stored subprograms, see Example 5–6 on page 5-11.

Creating a Stored Procedure With the AUTHID Clause

By default, stored procedures and functions execute with the privileges of their owner, not their current user. Such definer's rights subprograms are bound to the schema in which they reside, allowing you to refer to objects in the same schema without qualifying their names. For example, if schemas `HR` and `OE` both have a table called `departments`, a procedure owned by `HR` can refer to `departments` rather than the qualified `HR.departments`. If user `OE` calls the procedure owned by `HR`, the procedure still accesses the `departments` table owned by `HR`.

You can use the `AUTHID CURRENT_USER` clause to make stored procedures and functions execute with the privileges and schema context of the calling user. You can create one instance of the procedure, and many users can call it to access their own data because invoker's rights subprograms are not bound to a particular schema.

In Example 5–3, the procedure is created with the `AUTHID CURRENT_USER` clause. This example is based on Example 4–36 on page 4-28.

Example 5–3 Creating a Stored Procedure With the AUTHID Clause

```
CREATE OR REPLACE PROCEDURE create_log_table
-- use AUTHID CURRENT _USER to execute with the privileges and
-- schema context of the calling user
  AUTHID CURRENT_USER AS
  tabname       VARCHAR2(30); -- variable for table name
  temptabname   VARCHAR2(30); -- temporary variable for table name
  currentdate   VARCHAR2(8);  -- varible for current date
BEGIN
-- extract, format, and insert the year, month, and day from SYSDATE into
-- the currentdate variable
  SELECT TO_CHAR(EXTRACT(YEAR FROM SYSDATE)) ||
     TO_CHAR(EXTRACT(MONTH FROM SYSDATE),'FM09') ||
     TO_CHAR(EXTRACT(DAY FROM SYSDATE),'FM09') INTO currentdate FROM DUAL;
-- construct the log table name with the current date as a suffix
  tabname := 'log_table_' || currentdate;

-- check whether a table already exists with that name
-- if it does NOT exist, then go to exception handler and create table
-- if the table does exist, then note that table already exists
  SELECT TABLE_NAME INTO temptabname FROM USER_TABLES
    WHERE TABLE_NAME = UPPER(tabname);
  DBMS_OUTPUT.PUT_LINE('Table ' || tabname || ' already exists.');

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    -- this means the table does not exist because the table name
    -- was not found in USER_TABLES
      BEGIN
-- use EXECUTE IMMEDIATE to create a table with tabname as the table name
        EXECUTE IMMEDIATE 'CREATE TABLE ' || tabname
                         || '(op_time VARCHAR2(10), operation VARCHAR2(50))' ;
        DBMS_OUTPUT.PUT_LINE(tabname || ' has been created');
      END;

END create_log_table;
/

-- to call the create_log_table procedure, you can use the following
BEGIN
  create_log_table;
```

```
END;
/
```

For different methods to execute (call) stored subprograms, see Example 5–6 on page 5-11.

Creating Stored Functions With the SQL CREATE FUNCTION Statement

The SQL `CREATE FUNCTION` statement lets you create stored functions that are stored in an Oracle database. These stored (schema level) subprograms can be accessed from SQL. You can use the optional `OR REPLACE` clause to modify an existing function.

Example 5–4 is an example of a function that returns a character string that contains the upper case last and first names of an employee. The example also show how to run (call) the function.

Example 5–4 Creating a Stored Function That Returns a String

```
CREATE OR REPLACE FUNCTION last_first_name (empid NUMBER)
  RETURN VARCHAR2 IS
  lastname   employees.last_name%TYPE; -- declare a variable same as last_name
  firstname  employees.first_name%TYPE; -- declare a variable same as first_name
BEGIN
  SELECT last_name, first_name INTO lastname, firstname FROM employees
    WHERE employee_id = empid;
  RETURN ( 'Employee: ' || empid || ' - ' || UPPER(lastname)
                                 || ', ' || UPPER(firstname) );
END last_first_name;
/

-- you can use the following block to call the function
DECLARE
  empid NUMBER := 163; -- pick an employee ID to test the function
BEGIN
-- display the output of the function
  DBMS_OUTPUT.PUT_LINE( last_first_name(empid) );
END;
/

-- you can also call a function from a SQL SELECT statement
-- using the dummy DUAL table
SELECT last_first_name(163) FROM DUAL;
```

Example 5–5 is an example of a stored function that returns the calculated salary ranking for a specific employee based on the current minimum and maximum salaries of employees in the same job category.

Example 5–5 Creating a Stored Function That Returns a Number

```
-- function calculates the salary ranking of the employee based on the current
-- minimum and maximum salaries for employees in the same job category
CREATE OR REPLACE FUNCTION emp_sal_ranking (empid NUMBER)
  RETURN NUMBER IS
  minsal       employees.salary%TYPE; -- declare a variable same as salary
  maxsal       employees.salary%TYPE; -- declare a variable same as salary
  jobid        employees.job_id%TYPE; -- declare a variable same as job_id
  sal          employees.salary%TYPE; -- declare a variable same as salary
BEGIN
-- retrieve the jobid and salary for the specific employee ID
  SELECT job_id, salary INTO jobid, sal FROM employees WHERE employee_id = empid;
```

```
-- retrieve the minimum and maximum salaries for employees with the same job ID
  SELECT MIN(salary), MAX(salary) INTO minsal, maxsal FROM employees
      WHERE job_id = jobid;
-- return the ranking as a decimal, based on the following calculation
  RETURN ((sal - minsal)/(maxsal - minsal));
END emp_sal_ranking;
/

-- create a PL/SQL block to call the function, you can also use another subprogram
-- because a function returns a value, it is called as part of a line of code
DECLARE
  empid NUMBER := 163; -- pick an employee ID to test the function
BEGIN
-- display the output of the function, round to 2 decimal places
  DBMS_OUTPUT.PUT_LINE('The salary ranking for employee ' || empid || ' is: '
                       || ROUND(emp_sal_ranking(empid),2) );
END;
/
```

The output of the PL/SQL block is similar to:

```
The salary ranking for employee 163 is: .63
```

Calling Stored Procedures or Functions

You can call a stored subprogram from a `BEGIN` ... `END` block or from another subprogram or a package.

When calling a stored procedure or function, you can write the actual parameters using the following type of notation:

- Positional notation: You specify the same parameters in the same order as they are declared in the procedure. This notation is compact, but you must specify the parameters (especially literals) in the correct order.
- Named notation: You specify the name of each parameter and its value. An arrow (=>) serves as the association operator. The order of the parameters is not significant.
- Mixed notation: You specify the first parameters with positional notation, then switch to named notation for the last parameters.

Example 5–6 shows how you can call the stored procedure in Example 5–2.

Example 5–6 Techniques for Calling Stored Procedures or Functions

```
-- use a PL/SQL block to execute the procedure
BEGIN
  award_bonus(179, 0.05);
END;
/
-- using named notation for the parameters, rather than positional
BEGIN
  award_bonus(bonus_rate=>0.05, emp_id=>123);
END;
/
```

You can also call stored PL/SQL procedures and functions from Application Builder, Java programs, and PHP programs.

See Also:

- *Oracle Database Express Edition Application Express User's Guide* for information about calling stored PL/SQL procedures and functions from Application Builder
- *Oracle Database Express Edition 2 Day Plus Java Developer Guide* for information about calling stored PL/SQL procedures and functions from Java
- *Oracle Database Express Edition 2 Day Plus PHP Developer Guide* for information about calling stored PL/SQL procedures and functions from PHP

Editing Procedures or Functions

To edit procedures and functions, you can use the Object Browser page, the SQL Commands page, or the SQL `CREATE OR REPLACE` statement with SQL Command Line.

If you use the SQL `CREATE OR REPLACE` statement with SQL Command Line, you simply type in the modified procedure or function code. See "Entering and Executing SQL Statements and Commands" on page A-3.

To edit a procedure in the SQL Commands page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the home page, click the **SQL** icon to display the SQL page.
3. Click the **SQL Commands** icon to display the SQL Commands page.
4. Click the **Saved SQL** tab to display the saved SQL modules.
5. Click the name of the saved SQL that contains the procedure or function code that you want to edit.
6. Modify the source code for the procedure or function. Click the **Run** button if you want to execute the procedure or function.
7. When you are finished, you can click the **Save** button to save the code for future use.

To edit a subprogram with Object Browser:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user `HR` with your password for the `HR` account.
2. Click the **Object Browser** icon on the Database Home Page.

   The Object Browser home page appears.
3. Select **Procedures** or **Functions** in the object list, then click the subprogram you want to display.
4. With the subprogram displayed, click **Edit** button to modify the subprogram code.
5. Click the **Compile** button to ensure your changes did raise any errors when executed. Compiling the subprogram also saves the changes.

Dropping a Procedure or Function

You can drop a procedure or function from the database with the Object Browser page or the SQL `DROP` statement.

To use the Object Browser page to drop procedures and functions:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user `HR` with your password for the `HR` account.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the object list, select **Procedures** or **Functions**, then click the name of the procedure or function you want to drop.
4. Click the **Drop** button.
5. Click the **Finish** button to confirm the action.

To drop procedures or functions with SQL statements, use the SQL `DROP PROCEDURE` or `DROP FUNCTION` statement, as shown in Example 5–7.

Example 5–7 Dropping Subprograms With the DROP Statement

```
-- drop the procedure award_bonus to remove from the database
DROP PROCEDURE award_bonus;

-- drop the function emp_sal_ranking to remove from database
DROP FUNCTION emp_sal_ranking;
```

Managing Packages

You can create, modify, and drop packages and package bodies using the Object Browser page, the SQL Commands page, the Script Editor page, or SQL Command Line (SQL*Plus). You can view existing packages and package bodies with the Object Browser page.

The SQL `CREATE PACKAGE` statement is used to create package specification (specs). The `CREATE PACKAGE BODY` statement is used to define the package body.

See Also:

- See "CREATE PACKAGE" in *Oracle Database SQL Reference*
- See "CREATE PACKAGE BODY" in *Oracle Database SQL Reference*

This section contains the following topics:

- Writing Packages With PL/SQL Code on page 5-14
- Creating Packages in the SQL Commands Page on page 5-14
- Creating Packages With the Object Browser Page on page 5-15
- Viewing Packages With the Object Browser Page on page 5-16
- Creating Packages With the SQL CREATE PACKAGE Statement on page 5-16
- Editing Packages on page 5-18
- Dropping Packages on page 5-19
- Calling Procedures and Functions in Packages on page 5-20

- Accessing Variables in Packages on page 5-20
- Accessing Types in Packages on page 5-21

See Also: *Oracle Database Express Edition Application Express User's Guide* for information about managing packages with Object Browser

Writing Packages With PL/SQL Code

With PL/SQL, you can break down an application into well-defined modules. Using PL/SQL code, you can write program units that are stored as database objects that can be reused. These objects include packages, subprograms, and triggers. Subprograms and packages are discussed in this section; triggers are discussed in Chapter 6, "Using Triggers".

Guidelines for Writing Packages

When writing packages, keep them general so they can be reused in future applications. Become familiar with the Oracle-supplied packages, and avoid writing packages that duplicate features already provided by Oracle.

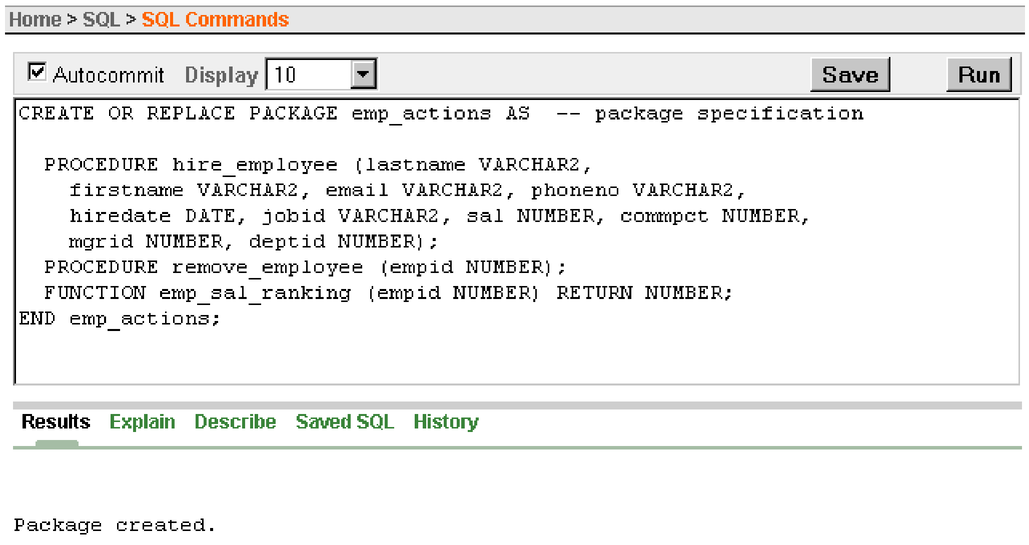
Design and define package specifications before the package bodies. Place in a specification only those parts that must be visible to calling programs. That way, other developers cannot build unsafe dependencies on your implementation details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package specification. Changes to a package body do not require recompiling calling procedures. Changes to a package specification require Oracle Database XE to recompile every stored subprogram that references the package.

Creating Packages in the SQL Commands Page

To create and run a package specification or body in the **SQL Commands** page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the home page, click the **SQL** icon to display the SQL page.
3. Click the **SQL Commands** icon to display the SQL Commands page.
4. On the SQL Commands page, enter the PL/SQL code for the package specification or body. Use the code in Example 5–8 on page 5-17.


Home > SQL > SQL Commands
Autocommit Display 10
Save
Run
Results Explain Describe Saved SQL History

```
CREATE OR REPLACE PACKAGE emp_actions AS  -- package specification

  PROCEDURE hire_employee (lastname VARCHAR2,
    firstname VARCHAR2, email VARCHAR2, phoneno VARCHAR2,
    hiredate DATE, jobid VARCHAR2, sal NUMBER, commpct NUMBER,
    mgrid NUMBER, deptid NUMBER);
  PROCEDURE remove_employee (empid NUMBER);
  FUNCTION emp_sal_ranking (empid NUMBER) RETURN NUMBER;
END emp_actions;
```

```
Package created.
```

5. Click the **Run** button to create the package specification or body. If necessary, select (highlight) only the specific code for creating the package specification or body before clicking the **Run** button. Any comments outside the package or package body block are not legal in the SQL Commands page.
6. If you want to save the PL/SQL code for future use, click the **Save** button.
7. In the **Name** field, enter a name for the saved PL/SQL code (`emp_actions_pkg_spec`). You can also enter an optional description. Click the **Save** button to save the code.
8. To access saved PL/SQL code, click the **Saved SQL** tab and select the name of the saved PL/SQL code that you want to access.
9. To create, run, and save the PL/SQL code for a package body, repeat the steps in this example with the code in Example 5–9 on page 5-17.

In the previous steps you created a package. For information about how to execute or call a subprogram in the package, see "Calling Procedures and Functions in Packages" on page 5-20.

> **See Also:** *Oracle Database Express Edition Application Express User's Guide* for detailed information about using SQL Scripts

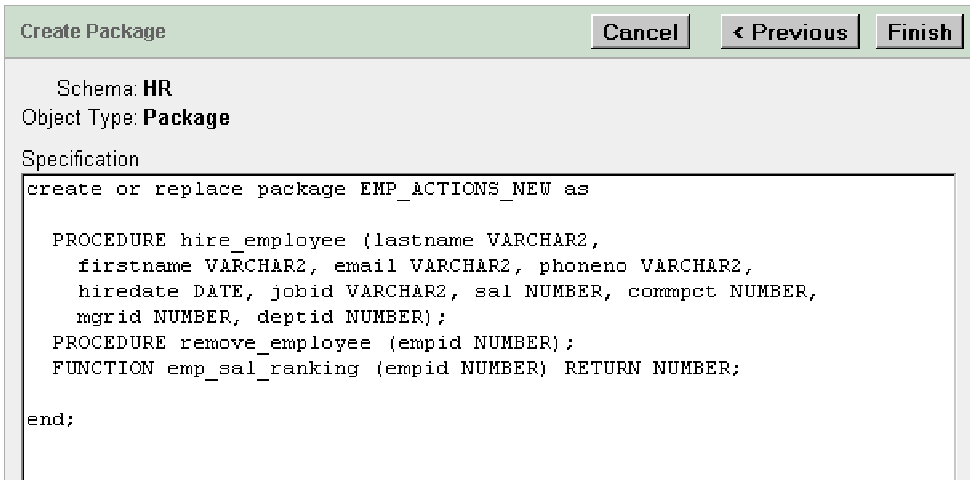
Creating Packages With the Object Browser Page

You can use the Object Browser page to create packages. This section explains how to create a package specification.

To create a package specification:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the Detail pane, select **Package** from the **Create** menu.
4. In the **Create Package** page, select the **Specification** option and click **Next**.
5. Enter the package name (`emp_actions_new`), and then click the **Next** button.

6. Enter the PL/SQL source code for the package specification. Use the code in Example 5–8 on page 5-17.



Schema: **HR**
Object Type: **Package**

Specification

```
create or replace package EMP_ACTIONS_NEW as

  PROCEDURE hire_employee (lastname VARCHAR2,
    firstname VARCHAR2, email VARCHAR2, phoneno VARCHAR2,
    hiredate DATE, jobid VARCHAR2, sal NUMBER, commpct NUMBER,
    mgrid NUMBER, deptid NUMBER);
  PROCEDURE remove_employee (empid NUMBER);
  FUNCTION emp_sal_ranking (empid NUMBER) RETURN NUMBER;

end;
```

7. After entering the code for the package specification, click the **Finish** button.
8. Click the **Body** tab, then the **Edit** button to enter the source code for the package body. Use the code in Example 5–9 on page 5-17, substituting `emp_actions_new` for `emp_actions`.
9. Click the **Compile** button to run the package. If errors are raised, correct the source code and try compiling again. Compiling the package also saves any changes made to the package.
10. When you have finished, click the **Finish** button.

In the previous steps, you created a package. For information about how to execute or call a subprogram in the package, see "Calling Procedures and Functions in Packages" on page 5-20.

Viewing Packages With the Object Browser Page

To find out which packages and package bodies exist in your database, use the Object Browser.

To use the Object Browser page to view packages and package bodies:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the object list, select **Packages** then click the name of the package you want to display.

   The package specification information displays.
4. With the package specification displayed, click the **Body** tab to view the package body if it exists.

Creating Packages With the SQL CREATE PACKAGE Statement

To create packages, use the SQL `CREATE PACKAGE` and `CREATE PACKAGE BODY` statements. You can use these SQL statements in the SQL Commands page, the Script Editor page, the Object Browser page, or SQL Command Line (SQL*Plus). In

Example 5–8 and Example 5–9, the `OR REPLACE` option is used so that you can update an existing package without having to first drop the package.

In Example 5–8, the `emp_actions` package specification contains two procedures that update the `employees` table and one function that provides information. The package specification provides the declaration of the subprograms. The package body provides the contents of the subprograms.

Example 5–8 Creating a Package Specification

```
CREATE OR REPLACE PACKAGE emp_actions AS  -- package specification

  PROCEDURE hire_employee (lastname VARCHAR2,
    firstname VARCHAR2, email VARCHAR2, phoneno VARCHAR2,
    hiredate DATE, jobid VARCHAR2, sal NUMBER, commpct NUMBER,
    mgrid NUMBER, deptid NUMBER);
  PROCEDURE remove_employee (empid NUMBER);
  FUNCTION emp_sal_ranking (empid NUMBER) RETURN NUMBER;
END emp_actions;
/
```

In Example 5–9, the `emp_actions` package body is created. The package body provides the contents of the subprograms in the package specification.

Example 5–9 Creating a Package Body

```
CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- package body

-- code for procedure hire_employee, which adds a new employee
  PROCEDURE hire_employee (lastname VARCHAR2,
    firstname VARCHAR2, email VARCHAR2, phoneno VARCHAR2, hiredate DATE,
    jobid VARCHAR2, sal NUMBER, commpct NUMBER, mgrid NUMBER, deptid NUMBER) IS
    min_sal    employees.salary%TYPE; -- variable to hold minimum salary for jobid
    max_sal    employees.salary%TYPE; -- variable to hold maximum salary for jobid
    seq_value  NUMBER;  -- variable to hold next sequence value
  BEGIN
    -- get the next sequence number in the employees_seq sequence
    SELECT employees_seq.NEXTVAL INTO seq_value FROM DUAL;
    -- use the next sequence number for the new employee_id
    INSERT INTO employees VALUES (seq_value, lastname, firstname, email,
     phoneno, hiredate, jobid, sal, commpct, mgrid, deptid);
     SELECT min_salary INTO min_sal FROM jobs WHERE job_id = jobid;
     SELECT max_salary INTO max_sal FROM jobs WHERE job_id = jobid;
     IF sal > max_sal THEN
       DBMS_OUTPUT.PUT_LINE('Warning: ' || TO_CHAR(sal)
                 || ' is greater than the maximum salary '
                 || TO_CHAR(max_sal) || ' for the job classification ' || jobid );
     ELSIF sal < min_sal THEN
       DBMS_OUTPUT.PUT_LINE('Warning: ' || TO_CHAR(sal)
                 || ' is less than the minimum salary '
                 || TO_CHAR(min_sal) || ' for the job classification ' || jobid );
     END IF;
  END hire_employee;

-- code for procedure remove_employee, which removes an existing employee
  PROCEDURE remove_employee (empid NUMBER) IS
     firstname employees.first_name%TYPE;
     lastname  employees.last_name%TYPE;
  BEGIN
    SELECT first_name, last_name INTO firstname, lastname FROM employees
```

```
      WHERE employee_id = empid;
    DELETE FROM employees WHERE employee_id = empid;
    DBMS_OUTPUT.PUT_LINE('Employee: ' || TO_CHAR(empid) || ', '
                        || firstname || ', ' || lastname || ' has been deleted.');
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee ID: ' || TO_CHAR(empid) || ' not found.');
  END remove_employee;

-- code for function emp_sal_ranking, which calculates the salary ranking of the
-- employee based on the minimum and maximum salaries for the job category
  FUNCTION emp_sal_ranking (empid NUMBER) RETURN NUMBER IS
    minsal        employees.salary%TYPE; -- declare a variable same as salary
    maxsal        employees.salary%TYPE; -- declare a variable same as salary
    jobid         employees.job_id%TYPE; -- declare a variable same as job_id
    sal           employees.salary%TYPE; -- declare a variable same as salary
  BEGIN
-- retrieve the jobid and salary for the specific employee ID
    SELECT job_id, salary INTO jobid, sal FROM employees
       WHERE employee_id = empid;
-- retrieve the minimum and maximum salaries for the job ID
    SELECT min_salary, max_salary INTO minsal, maxsal FROM jobs
       WHERE job_id = jobid;
-- return the ranking as a decimal, based on the following calculation
    RETURN ((sal - minsal)/(maxsal - minsal));
  END emp_sal_ranking;
END emp_actions;
/

-- the following BEGIN..END block calls, or executes, the emp_sal_ranking
-- function in the emp_actions package with an argument value
DECLARE
  empid NUMBER := 163; -- use a test value for the employee_id
BEGIN
  DBMS_OUTPUT.put_line('The salary ranking for employee ' || empid || ' is: '
                        || ROUND(emp_actions.emp_sal_ranking(empid),2) );
END;
/
```

The output of the PL/SQL block is similar to:

```
The salary ranking for employee 163 is: .58
```

Note that the function result for employee 163 is different from the result for Example 5–5 on page 5-10. While the functions have the same function name (`emp_sal_ranking`), they are not the same function. The function in the package is identified by the package name prefix, as in `emp_actions.emp_sal_ranking`.

For methods on calling subprograms in a package, see "Calling a Subprogram in a Package" on page 5-20.

Editing Packages

To edit packages and package bodies, you can use the Object Browser page, the SQL Commands page, or the SQL `CREATE OR REPLACE` statement with SQL Command Line.

If you use the SQL `CREATE OR REPLACE` statement with SQL Command Line, you simply type in the modified package specification or body code. See "Entering and Executing SQL Statements and Commands" on page A-3.

To edit a package in the SQL Commands page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. On the home page, click the **SQL** icon to display the SQL page.
3. Click the **SQL Commands** icon to display the SQL Commands page.
4. Click the **Saved SQL** tab to display the saved SQL modules.
5. Click the name of the saved SQL that contains the package code that you want to edit.
6. Modify the source code for the package. Click the **Run** button if you want to execute the package.
7. When you are finished, you can click the **Save** button to save the code for future use.

To edit a package with the Object Browser page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user HR with your password for the HR account.
2. On the Database Home Page, click the **Object Browser** icon.
3. In the object list, select **Packages** and then click the package you want to display.

   The package specification information displays.
4. With the package specification displayed, click the **Edit** button to modify the package specification. You can click the **Body** tab to edit the source code for the package body if it exists.
5. Click the **Compile** button to ensure your changes did raise any errors when executed. Compiling the package also saves the changes.

Dropping Packages

You can use the SQL DROP statement or the Object Browser page to drop packages and package bodies.

You can drop a package or package body with the SQL DROP statement. When drop a package specification, the corresponding package body is dropped also. You can choose to drop only the package body. For example:

```
-- drop only the package body
DROP PACKAGE BODY my_package;
-- drop the package specification and package body
DROP PACKAGE my_package;
```

To drop a package or package body with the Object Browser page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user HR with your password for the HR account.
2. On the Database Home Page, click the **Object Browser** icon.
3. Select **Packages** in the object list, then click the package you want to display.

   The package specification information displays.

4. With the package specification displayed, click the **Drop** button to drop the package specification and package body. You can click the **Body** tab and then the **Drop** button to drop only the packaged body if it exists.
5. Click the **Finish** button to confirm that you want to drop the package specification or package body.

Calling Procedures and Functions in Packages

To call the procedures or functions of the `emp_actions` package created in Example 5–9, you can execute the statements in Example 5–10. The subprograms can be executed in a `BEGIN` .. `END` block or from another subprogram. Note the use of the package name as a prefix to the subprogram name.

Example 5–10 Calling a Subprogram in a Package

```
-- the following calls the hire_employee subprogram in the emp_actions package
-- with the associated parameter values
BEGIN
  emp_actions.hire_employee('Townsend', 'Mark', 'MTOWNSEND',
   '555.123.2222', '31-JUL-05', 'AC_MGR', 9000, .1, 101, 110);
END;
/

-- the following calls the remove_employee subprogram in the emp_actions package
-- in this case, remove the employee just added (employee_id = 208)
-- note that the employee ID might be different on your system
BEGIN
  emp_actions.remove_employee(208);
END;
/

-- cleanup: drop the package
DROP PACKAGE emp_actions;
```

Packages are stored in the database, where they can be shared by many applications. Calling a packaged subprogram for the first time loads the whole package and caches it in memory, saving on disk I/O for subsequent calls. Thus, packages enhance reuse and improve performance in a multiple-user, multiple-application environment.

If a subprogram does not take any parameters, you can include an empty set of parentheses or omit the parentheses, both in PL/SQL and in functions called from SQL queries. For calls to a method that takes no parameters, an empty set of parentheses is optional within PL/SQL scopes, but they are required within SQL scopes.

Accessing Variables in Packages

You can create a package specification that is designated only to supply common variables to other packages or subprograms. With the variables in one package, they can be easily maintained for all subprograms that use the variables, rather than maintaining the variables in all the individual subprograms. Common variables are typically used in multiple subprograms, such as a sales tax rate.

In Example 5–11, the variables `my_var_pi`, `my_var_e`, and `my_var_sales_tax` can be used by any subprogram. If you change the value of any of those variables, then all subprograms that use the variable will get the new value without having to change anything in those individual subprograms.

Note that you need to use of the package name as a prefix to the variable name, such as `my_var_pkg.my_var_pi`.

Example 5–11 Creating Variables in a PL/SQL Package Specification

```
CREATE OR REPLACE PACKAGE my_var_pkg AS
-- set up a variable for pi, used in calculations with circles and spheres
  my_var_pi         NUMBER := 3.14016408289008292431940027343666863227;
-- set up a variable for e, the base of the natural logarithm
  my_var_e          NUMBER := 2.71828182845904523536028747135266249775;
-- set up a variable for the current retail sales tax rate
  my_var_sales_tax  NUMBER := 0.0825;
END my_var_pkg;
/
```

Example 5–12 shows how variables that are defined in the `my_var_pkg` package specification can be used in PL/SQL subprograms.

Example 5–12 Using Variables From a Package Specification

```
CREATE OR REPLACE PROCEDURE circle_area(radius NUMBER) IS
  c_area NUMBER;
BEGIN
-- the following uses the value of the my_var_pi variable in my_var_pkg for pi
-- in the following calculation of the area of a circle
  c_area := my_var_pkg.my_var_pi * radius**2;
  DBMS_OUTPUT.PUT_LINE('Radius: ' || TO_CHAR(radius)
                       || ' Area: ' || TO_CHAR(c_area) );
END circle_area;
/

BEGIN -- some examples of the use of package variables
-- call the circle_area procedure with radius equal to 3, my_var_pi is used to
-- calculate the area in circle_area
  circle_area(3);
-- determine the sales tax on a $25 item using my_var_sales_tax for the tax rate
  DBMS_OUTPUT.PUT_LINE('Sales tax on $25.99 is $'
                        || TO_CHAR(25.99 * my_var_pkg.my_var_sales_tax) );
END;
/
```

Accessing Types in Packages

You can create a package specification that is designated only to supply common types, along with common variables, to other packages or subprograms. With the types in one package, they can be easily maintained for all subprograms that use the types, rather than maintaining the types in all the individual subprograms. Common types, such as a `REF CURSOR`, can be used to declare variables in other packages and subprograms. See "Cursor Variables (REF CURSORs)" on page 4-22.

In Example 5–13, the `emp_refcur_typ` and `my_refcur_typ` types can be used by any subprogram to declare cursor variables. Note that you need to use of the package name as a prefix to the type name, such as `my_var_pkg.my_refcur_typ`.

Example 5–13 Creating Types and Variables in a PL/SQL Package Specification

```
CREATE OR REPLACE PACKAGE my_var_pkg AS
-- set up a strongly typed cursor variable for the employees table
  TYPE emp_refcur_typ IS REF CURSOR RETURN employees%ROWTYPE;
```

```
-- set up a weakly typed cursor variable for multiple use
  TYPE my_refcur_typ IS REF CURSOR;
-- set up a variable for pi, used in calculations with circles and spheres
  my_var_pi         NUMBER := 3.14016408289008292431940027343666863227;
-- set up a variable for e, the base of the natural logarithm
  my_var_e          NUMBER := 2.71828182845904523536028747135266249775;
-- set up a variable for the current retail sales tax rate
  my_var_sales_tax  NUMBER := 0.0825;
END my_var_pkg;
/
```

Example 5–14 show how the `emp_refcur_typ` cursor variable that is defined in the `my_var_pkg` package specification can be used in PL/SQL subprograms.

Example 5–14 Using the emp_refcur_typ REF CURSOR From a Package Specification

```
-- this procedure uses the strongly-typed my_var_pkg.emp_refcur_typ REF CURSOR
CREATE OR REPLACE PROCEDURE display_emp_cursor (
                             emp_cursor IN OUT my_var_pkg.emp_refcur_typ) AS
  person employees%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('-- Here are the employees in the result set --');
  LOOP
    FETCH emp_cursor INTO person;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(person.employee_id || ' - ' || person.last_name
                                            || ', ' || person.first_name);
  END LOOP;
END display_emp_cursor;
/

-- this procedure uses the strongly-typed my_var_pkg.emp_refcur_typ REF CURSOR
CREATE OR REPLACE PROCEDURE get_emp_id (firstname IN VARCHAR2,
                                        lastname IN VARCHAR2) AS
  emp_cursor my_var_pkg.emp_refcur_typ;
BEGIN
-- search for employee IDs based on the input for first and last names
  OPEN emp_cursor FOR SELECT * FROM employees
    WHERE SUBSTR(UPPER(first_name), 1, LENGTH(firstname)) = UPPER(firstname)
    AND SUBSTR(UPPER(last_name), 1, LENGTH(lastname)) = UPPER(lastname);
-- pass emp_cursor to the display_emp_cursor procedure for processing
  display_emp_cursor(emp_cursor);
  CLOSE emp_cursor;
END get_emp_id;
/

BEGIN -- some examples of the use of package types
-- call the get_emp_id procedure that uses a REF CURSOR defined in a package
  get_emp_id('steve', 'kin');
END;
/
```

Example 5–15 show how the `my_refcur_typ` cursor variable that is defined in the `my_var_pkg` package specification can be used to return a result set that could be accessed by other subprograms.

Example 5–15 Using the my_refcur_typ REF CURSOR From a Package Specification

```
-- this procedure uses the weakly-typed my_var_pkg.my_refcur_typ REF CURSOR
CREATE OR REPLACE PROCEDURE get_emp_info (firstname IN VARCHAR2,
  lastname IN VARCHAR2, emp_cursor IN OUT my_var_pkg.my_refcur_typ) AS
BEGIN
-- the following returns employee info based on first and last names
  OPEN emp_cursor FOR SELECT employee_id, first_name, last_name, email,
    phone_number FROM employees
    WHERE SUBSTR(UPPER(first_name), 1, LENGTH(firstname)) = UPPER(firstname)
    AND SUBSTR(UPPER(last_name), 1, LENGTH(lastname)) = UPPER(lastname);
END get_emp_info;
/

-- the procedure can be updated to change the columns returned in the result set
CREATE OR REPLACE PROCEDURE get_emp_info (firstname IN VARCHAR2,
  lastname IN VARCHAR2, emp_cursor IN OUT my_var_pkg.my_refcur_typ) AS
BEGIN
-- because this procedure uses a weakly typed REF CURSOR, the cursor is flexible
-- and the SELECT statement can be changed, as in the following
  OPEN emp_cursor FOR SELECT e.employee_id, e.first_name, e.last_name, e.email,
    e.phone_number, e.hire_date, j.job_title FROM employees e
    JOIN jobs j ON e.job_id = j.job_id
    WHERE SUBSTR(UPPER(first_name), 1, LENGTH(firstname)) = UPPER(firstname)
    AND SUBSTR(UPPER(last_name), 1, LENGTH(lastname)) = UPPER(lastname);
END get_emp_info;
/
```

Oracle Provided Packages

Oracle Database XE provides product-specific packages that define application programming interfaces (APIs) you can call from PL/SQL, SQL, Java, or other programming environments. This section includes a list of the most common packages with a brief description and an overview of a few useful packages.

This section contains the following topics:

- List of Oracle Database XE Packages on page 5-23
- Overview of Some Useful Packages on page 5-27

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about and usage of provided packages

List of Oracle Database XE Packages

Table 5–1 provides a list of the common PL/SQL packages included with Oracle Database XE.

Table 5–1 Summary of Oracle Supplied PL/SQL Packages

Package Name	Description
DBMS_ALERT	Provides support for the asynchronous notification of database events.
DBMS_APPLICATION_INFO	Lets you register an application name with the database for auditing or performance tracking purposes.
DBMS_CHANGE_NOTIFICATION	Is part of a set of features that clients use to receive notifications when result sets of a query have changed. The package contains interfaces that can be used by mid-tier clients to register objects and specify delivery mechanisms.

Table 5–1 (Cont.) Summary of Oracle Supplied PL/SQL Packages

Package Name	Description
DBMS_CRYPTO	Lets you encrypt and decrypt stored data, can be used in conjunction with PL/SQL programs running network communications, and supports encryption and hashing algorithms.
DBMS_DATAPUMP	Lets you move all, or part of, a database between databases, including both data and metadata.
DBMS_DB_VERSION	Specifies the Oracle version numbers and other information useful for simple conditional compilation selections based on Oracle versions.
DBMS_DDL	Provides access to some SQL DDL statements from stored procedures, and provides special administration operations not available as DDLs.
DBMS_DEBUG	Implements server-side debuggers and provides a way to debug server-side PL/SQL program units.
DBMS_DESCRIBE	Describes the arguments of a stored procedure with full name translation and security checking.
DBMS_EPG	Implements the embedded PL/SQL gateway that enables a Web browser to invoke a PL/SQL stored procedure through an HTTP listener.
DBMS_ERRLOG	Provides a procedure that enables you to create an error logging table so that DML operations can continue after encountering errors rather than abort and roll back.
DMBS_FILE_TRANSFER	Lets you copy a binary file within a database or to transfer a binary file between databases.
DBMS_JOB	Lets you schedule administrative procedures that you want performed at periodic intervals; it is also the interface for the job queue.
DBMS_LOCK	Lets you request, convert and release locks through Oracle Lock Management services.
DBMS_METADATA	Lets callers easily retrieve complete database object definitions (metadata) from the dictionary.
DBMS_OBFUSCATION_TOOLKIT	Provides procedures for Data Encryption Standards.
DBMS_OUTPUT	Displays output from stored procedures, packages, and triggers, which is especially useful for displaying PL/SQL debugging information.
DBMS_PIPE	Provides a DBMS pipe service which enables messages to be sent between sessions.
DBMS_RANDOM	Provides a built-in random number generator.
DBMS_RESUMABLE	Lets you suspend large operations that run out of space or reach space limits after executing for a long time, fix the problem, and make the statement resume execution.
DBMS_ROWID	Provides procedures to create rowids and to interpret their contents.
DBMS_SCHEDULER	Provides a collection of scheduling functions that are callable from any PL/SQL program.
DBMS_SERVER_ALERT	Lets you issue alerts when some threshold has been violated.
DBMS_SESSION	Provides access to SQL `ALTER SESSION` statements, and other session information, from stored procedures.

Table 5–1 (Cont.) Summary of Oracle Supplied PL/SQL Packages

Package Name	Description
`DBMS_SQL`	Lets you use dynamic SQL to access the database.
`DBMS_TDB`	Reports whether a database can be transported between platforms using the RMAN `CONVERT DATABASE` command. It verifies that databases on the current host platform are of the same endian format as the destination platform, and that the state of the current database does not prevent transport of the database.
`DBMS_TTS`	Checks if the transportable set is self-contained.
`DBMS_TYPES`	Consists of constants, which represent the built-in and user-defined types.
`DBMS_UTILITY`	Provides various utility routines.
`DBMS_WARNING`	Provides the interface to query, modify and delete current system or session settings.
`DBMS_XDB`	Describes Resource Management and Access Control APIs for PL/SQL
`DBMS_XDB_VERSION`	Describes versioning APIs
`DBMS_XDBT`	Describes how an administrator can create a ConText index on the XML DB hierarchy and configure it for automatic maintenance
`DBMS_XDBZ`	Controls the Oracle XML DB repository security, which is based on Access Control Lists (ACLs).
`DBMS_XMLDOM`	Explains access to XMLType objects
`DBMS_XMLGEN`	Converts the results of a SQL query to a canonical XML format.
`DBMS_XMLPARSER`	Explains access to the contents and structure of XML documents.
`DMBS_XMLQUERY`	Provides database-to-XMLType functionality.
`DBMS_XMLSAVE`	Provides XML-to-database-type functionality.
`DBMS_XMLSCHEMA`	Explains procedures to register and delete XML schemas.
`DBMS_XMLSTORE`	Provides the ability to store XML data in relational tables.
`DBMS_XPLAN`	Describes how to format the output of the `EXPLAIN PLAN` command.
`DBMS_XSLPROCESSOR`	Explains access to the contents and structure of XML documents.
`HTF`	Hypertext functions generate HTML tags.
`HTMLDB_APPLICATION`	Enables users to take advantage of global variables
`HTMLDB_CUSTOM_AUTH`	Enables users to create form elements dynamically based on a SQL query instead of creating individual items page by page.
`HTMLDB_ITEM`	Enables users to create form elements dynamically based on a SQL query instead of creating individual items page by page.

Table 5–1 (Cont.) Summary of Oracle Supplied PL/SQL Packages

Package Name	Description
`HTMLDB_UTIL`	Provides utilities for getting and setting session state, getting files, checking authorizations for users, resetting different states for users, and also getting and setting preferences for users.
`HTP`	Hypertext procedures generate HTML tags.
`OWA_CACHE`	Provides an interface that enables the `PL/SQL Gateway` cache to improve the performance of PL/SQL Web applications.
`OWA_COOKIE`	Provides an interface for sending and retrieving HTTP cookies from the client's browser.
`OWA_CUSTOM`	Provides a Global PLSQL Agent Authorization callback function
`OWA_IMAGE`	Provides an interface to access the coordinates where a user clicked on an image.
`OWA_OPT_LOCK`	Contains subprograms that impose optimistic locking strategies so as to prevent lost updates.
`OWA_PATTERN`	Provides an interface to locate text patterns within strings and replace the matched string with another string.
`OWA_SEC`	Provides an interface for custom authentication.
`OWA_TEXT`	Contains subprograms used by `OWA_PATTERN` for manipulating strings. They are externalized so you can use them directly.
`OWA_UTIL`	Contains utility subprograms for performing operations such as getting the value of CGI environment variables, printing the data that is returned to the client, and printing the results of a query in an HTML table.
`UTL_COLL`	Enables PL/SQL programs to use collection locators to query and update.
`UTL_COMPRESS`	Provides a set of data compression utilities.
`UTL_DBWS`	Provides database Web services.
`UTL_ENCODE`	Provides functions that encode RAW data into a standard encoded format so that the data can be transported between hosts.
`UTL_FILE`	Enables your PL/SQL programs to read and write operating system text files and provides a restricted version of standard operating system stream file I/O.
`UTL_HTTP`	Enables HTTP callouts from PL/SQL and SQL to access data on the Internet or to call Oracle Web Server Cartridges.
`UTL_I18N`	Provides a set of services (Oracle Globalization Service) that help developers build multilingual applications.
`UTL_INADDR`	Provides a procedure to support internet addressing.
`UTL_LMS`	Retrieves and formats error messages in different languages.
`UTL_MAIL`	A utility for managing e-mail which includes commonly used e-mail features, such as attachments, CC, BCC, and return receipt.

Table 5–1 (Cont.) Summary of Oracle Supplied PL/SQL Packages

Package Name	Description
`UTL_RAW`	Provides SQL functions for `RAW` datatypes that concat, substr to and from `RAWS`.
`UTL_RECOMP`	Recompiles invalid PL/SQL modules, Java classes, indextypes and operators in a database, either sequentially or in parallel.
`UTL_REF`	Enables a PL/SQL program to access an object by providing a reference to the object.
`UTL_SMTP`	Provides PL/SQL functionality to send e-mails.
`UTL_TCP`	Provides PL/SQL functionality to support simple TCP/IP-based communications between servers and the outside world.
`UTL_URL`	Provides escape and unescape mechanisms for URL characters.

Overview of Some Useful Packages

This section provides a summary of some useful packages.

This section contains the following topics:

- DBMS_OUTPUT Package on page 5-27
- DBMS_RANDOM Package on page 5-27
- HTP Package on page 5-28
- UTL_FILE Package on page 5-30

DBMS_OUTPUT Package

The `DBMS_OUTPUT` package enables you to display output from PL/SQL blocks, subprograms, packages, and triggers. This package is especially useful for displaying PL/SQL debugging information. The `PUT_LINE` procedure outputs information to a buffer that can be read by another trigger, procedure, or package. You display the information by calling the `GET_LINE` procedure or by setting the `SERVEROUTPUT ON` setting in SQL Command Line.

For more information, see "Inputting and Outputting Data with PL/SQL" on page 4-5. For examples of the use of `DBMS_OUTPUT.PUT_LINE`, see Example 5–1 on page 5-7, Example 5–2 on page 5-7, and Example 5–3 on page 5-9.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_OUTPUT` packages

DBMS_RANDOM Package

The `DBMS_RANDOM` package provides a built-in random number generator. `DBMS_RANDOM` can be explicitly initialized, but does not need to be initialized before calling the random number generator. It will automatically initialize with the date, userid, and process id if no explicit initialization is performed.

If this package is seeded twice with the same seed, then accessed in the same way, it will produce the same results in both cases.

The `DBMS_RANDOM.VALUE` function can be called with no parameters to return a random number, greater than or equal to 0 and less than 1, with 38 digits to the right of the decimal (38-digit precision). Alternatively, you can call the function with low and high parameters to return a random number which is greater than or equal to the low parameter and less than high parameter.

Example 5–16 shows the use of the `DBMS_RANDOM.VALUE` function to return random numbers from 1 ton 100. The random numbers are truncated to integer values and stored in an array.

Example 5–16 Using the DBMS_RANDOM Package

```
DECLARE
-- declare an array type with 10 elements of NUMBER
  TYPE random_array IS VARRAY(10) OF NUMBER;
  random_numbers random_array;
  j NUMBER;
BEGIN
  random_numbers := random_array(); -- initialize the array
  FOR i IN 1..10 LOOP
-- add an element to the array
    random_numbers.EXTEND(1);
-- insert a random number in the next element in the array
    random_numbers(i) := TRUNC(DBMS_RANDOM.VALUE(1,101));

    j := 1;
-- make sure the random number is not already in the array
-- if it is, generated a new random number and check again
    WHILE j < random_numbers.LAST LOOP
       IF random_numbers(i) = random_numbers(j) THEN
          random_numbers(i) := TRUNC(DBMS_RANDOM.VALUE(1,101));
          j := 1;
       ELSE
         j := j + 1;
       END IF;
    END LOOP;

 END LOOP;

-- display the random numbers in the array
 FOR k IN random_numbers.FIRST..random_numbers.LAST LOOP
   DBMS_OUTPUT.PUT_LINE(random_numbers(k));
 END LOOP;

END;
/
```

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_RANDOM` packages

HTP Package

With the `HTP` package, you can create a Web page using `HTP` hypertext procedures to generate HTML tags. For example the procedure `HTP.PARA` generates the `<P>` paragraph tag and `HTP.ANCHOR` generates the `<A>` anchor tag. You can also use `HTP.PRINT` to explicit print HTML tags.

Note that for nearly every `HTP` procedure that generates one or more HTML tags, there is a corresponding `HTF` package hypertext function with identical parameters.

Example 5–17 is a modification of Example 4–29 on page 4-21 using the `HTP.PRINT` procedure. For each `DBMS_OUTPUT.PUT_LINE` in the original example, an `HTP.PRINT` has been substituted in the modified example.

Example 5–17 Using HTP Print Procedure

```
CREATE OR REPLACE PROCEDURE htp_yearly_bonus AS
-- declare variables for data fetched from cursor
  empid       employees.employee_id%TYPE; -- variable for employee_id
  hiredate    employees.hire_date%TYPE;   -- variable for hire_date
  firstname   employees.first_name%TYPE;  -- variable for first_name
  lastname    employees.last_name%TYPE;   -- variable for last_name
  rowcount    NUMBER;
  bonusamount NUMBER;
  yearsworked NUMBER;
-- declare the cursor with a parameter
  CURSOR cursor1 (thismonth NUMBER)IS
    SELECT employee_id, first_name, last_name, hire_date FROM employees
       WHERE EXTRACT(MONTH FROM hire_date) = thismonth;
BEGIN
  HTP.PRINT('<html>');                                  -- HTML open
  HTP.PRINT('<head>');                                  -- HEAD open
  HTP.PRINT('<title>Using the HTP Package</title>');    -- title line
  HTP.PRINT('</head>');                                 -- HEAD close
  HTP.PRINT('<body TEXT="#000000" BGCOLOR="#FFFFFF">') ; -- BODY open
-- open and pass a parameter to cursor1, select employees hired on this month
  OPEN cursor1(EXTRACT(MONTH FROM SYSDATE));
  HTP.PRINT('<h1>----- Today is ' || TO_CHAR(SYSDATE, 'DL') || ' -----</h1>');
  HTP.PRINT('<p>Employees with yearly bonus amounts:</p>');
  HTP.PRINT('<pre>'); -- insert the preformat tag
  LOOP
-- fetches 4 columns into variables
    FETCH cursor1 INTO empid, firstname, lastname, hiredate;
-- check the cursor attribute NOTFOUND for the end of data
    EXIT WHEN cursor1%NOTFOUND;
-- calculate the yearly bonus amount based on months (years) worked
  yearsworked := ROUND( (MONTHS_BETWEEN(SYSDATE, hiredate)/12) );
  IF yearsworked > 10   THEN bonusamount := 2000;
    ELSIF yearsworked > 8 THEN bonusamount := 1600;
    ELSIF yearsworked > 6 THEN bonusamount := 1200;
    ELSIF yearsworked > 4 THEN bonusamount := 800;
    ELSIF yearsworked > 2 THEN bonusamount := 400;
    ELSIF yearsworked > 0 THEN bonusamount := 100;
  END IF;
-- display the employee Id, first name, last name, hire date, and bonus
-- for each record (row) fetched
    HTP.PRINT( empid || ' ' || RPAD(firstname, 21, ' ') ||
         RPAD(lastname, 26, ' ') || hiredate || TO_CHAR(bonusamount, '$9,999'));
  END LOOP;
  HTP.PRINT('</pre>'); -- end the preformat tag
  rowcount := cursor1%ROWCOUNT;
  HTP.PRINT('<p>The number of rows fetched is ' || rowcount || '</p>');
  CLOSE cursor1;
  HTP.PRINT('</body>');                                 -- BODY close
  HTP.PRINT('</html>');                                 -- HTML close
END;
/
```

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about the `HTP` packages

UTL_FILE Package

The UTL_FILE package enables PL/SQL programs to read and write operating system (OS) text files. It provides a restricted version of standard OS stream file I/O, including open, put, get, and close operations. When you want to read or write a text file, you call the FOPEN function, which returns a file handle for use in subsequent procedure calls. When opening a file with FOPEN, the file can be opened in append (A), read (R), or write (W) mode. After a file is opened, you can use UTL_FULE procedures such as PUT_LINE to write a text string and line terminator to an open file and GET_LINE to read a line of text from an open file into an output buffer.

Before a user can run UTL_FILE procedures, the user must be granted access to UTL_FILE and there must be an accessible directory for the user to read and write files. As the user SYS, you need to run the SQL GRANT EXECUTE statement to provide access to the UTL_FILE package, run the SQL CREATE DIRECTORY statement to set up an accessible directory, and run the SQL GRANT ... DIRECTORY statement to grant privileges to that directory. Example 5–18 shows how to set up an existing directory and grant the HR user access to that directory.

Example 5–18 Setting up a Directory for Use With UTL_FILE

```
-- first connect as SYS to perform the necessary setups
-- when you run the following to connect as SYS, use your password for SYS
CONNECT SYS/ORACLE AS SYSDBA
-- the following grants access on the UTL_FILE package to user HR
GRANT EXECUTE ON UTL_FILE TO HR;
-- the following sets up directory access for /tmp on a Linux platform
CREATE OR REPLACE DIRECTORY temp_dir AS '/tmp';
-- you could use 'c:\temp' for temp_dir on a Windows platform, note that
-- c:\temp must exist on the Windows computer
-- the following grants the user read and write access to the directory
GRANT READ, WRITE ON DIRECTORY temp_dir TO HR;
-- now connect as user HR/HR to check directory setup
-- when you connect as HR, use your password for HR
CONNECT HR/HR
-- the following SELECT query lists information about all directories that
-- have been set up for the user
SELECT * FROM ALL_DIRECTORIES;
-- if TEMP_DIR is listed, then you are ready to run UTL_FILE procedures as HR
```

After the SQL statements in Example 5–18 are executed, you can connect as the user HR and run UTL_FILE procedures. Some simple examples are shown in Example 5–19.

Example 5–19 Using the UTL_FILE Package

```
-- connect as user HR and run UTL_FILE procedures
DECLARE
  string1 VARCHAR2(32767);
  file1 UTL_FILE.FILE_TYPE;
BEGIN
  file1 := UTL_FILE.FOPEN('TEMP_DIR','log_file_test','A'); -- open in append mode
  string1 := TO_CHAR(SYSDATE) || ' UTL_FILE test';
  UTL_FILE.PUT_LINE(file1, string1); -- write a string to the file
  UTL_FILE.FFLUSH(file1);
  UTL_FILE.FCLOSE_ALL; -- close all open files
END;
/

DECLARE
  string1 VARCHAR2(32767);
  file1 UTL_FILE.FILE_TYPE;
BEGIN
  file1 := UTL_FILE.FOPEN('TEMP_DIR','log_file_test','R'); -- open in read mode
  UTL_FILE.GET_LINE(file1, string1, 32767); -- read a string from the file
  DBMS_OUTPUT.PUT_LINE(string1); -- display the string
  UTL_FILE.FCLOSE_ALL; -- close all open files
END;
/
```

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about the `UTL_FILE` packages

6 Using Triggers

This section discusses the development of triggers with PL/SQL code and the use of database triggers with Oracle Database Express Edition.

This section contains the following topics:

See Also:

- *Oracle Database Concepts* for conceptual information about triggers
- *Oracle Database Application Developer's Guide - Fundamentals* for information about coding triggers
- *Oracle Database SQL Reference* for information about the `CREATE TRIGGER` SQL statement

Overview of Triggers

A database trigger is a stored procedure associated with a database table, view, or event. The trigger can be called once, when some event occurs, or many times, once for each row affected by an `INSERT`, `UPDATE`, or `DELETE` statement. The trigger can be called after the event, to record it, or take some follow-up action. The trigger can be called before the event, to prevent erroneous operations or fix new data so that it conforms to business rules. The executable part of a trigger can contain procedural statements and SQL data manipulation statements.

Triggers are created using the SQL `CREATE TRIGGER` statement. This statement can be used with Object Browser, SQL Script Editor, or SQL Command Line (SQL*Plus). The `CREATE` (or `CREATE OR REPLACE`) statement fails if any errors exist in the PL/SQL block.

This section contains the following topics:

- Enabled and Disabled Trigger Modes on page 6-5
- Error Conditions and Exceptions in the Trigger Body on page 6-5

See Also: *Oracle Database SQL Reference* for information about trigger creation syntax

Types of Triggers

A trigger can be a stored PL/SQL or C procedure associated with a table, view, schema, or the database itself. Oracle Database XE automatically executes a trigger when a specified event takes place, which usually is a DML statement being issued against the table. The types of triggers are:

- DML triggers on tables
- `INSTEAD OF` triggers on views
- System triggers on `DATABASE` or `SCHEMA`

You can create triggers to be fired on any of the following:

- DML statements (`DELETE`, `INSERT`, `UPDATE`)
- DDL statements (`CREATE`, `ALTER`, `DROP`)
- Database operations (`LOGON`, `LOGOFF`)

Naming Triggers

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need to be unique with respect to other schema objects, such as tables, views, and procedures. For example, a table and a trigger can have the same name; however, to avoid confusion, this is not recommended.

When Is a Trigger Fired?

A trigger is fired based on a triggering statement, which specifies:

- The SQL statement or the system event, database event, or DDL event that fires the trigger body. The options include `DELETE`, `INSERT`, and `UPDATE`. One, two, or all three of these options can be included in the triggering statement specification.
- The table, view, database, or schema associated with the trigger.

If a trigger contained the following statement:

```
AFTER DELETE OR INSERT OR UPDATE ON employees ...
```

then any of the following statements would fire the trigger:

```
DELETE FROM employees WHERE ...;
INSERT INTO employees VALUES ( ... );
INSERT INTO employees SELECT ... FROM ... ;
UPDATE employees SET ... ;
```

An `UPDATE` statement might include a list of columns. If a triggering statement includes a column list, the trigger is fired only when one of the specified columns is updated. If a triggering statement omits a column list, the trigger is fired when any column of the associated table is updated. A column list cannot be specified for `INSERT` or `DELETE` triggering statements. In Example 6–1 on page 6-11 the `audit_sal` trigger specifies the `salary` column, and is only fired after an `UPDATE` of the salary of an employee in the `employees` table. Updates of other columns would not fire the trigger.

Controlling When a Trigger Is Fired

This section describes options that control when a trigger is fired.

This section contains the following topics:

- Firing Triggers With the BEORE and AFTER Options on page 6-3
- Firing Triggers With the FOR EACH ROW Option on page 6-3
- Firing Triggers Based on Conditions (WHEN Clause) on page 6-4
- Firing Triggers With the INSTEAD OF Option on page 6-4

Firing Triggers With the BEORE and AFTER Options

The `BEFORE` or `AFTER` option in the `CREATE TRIGGER` statement specifies exactly when to fire the trigger body in relation to the triggering statement that is being run. In a `CREATE TRIGGER` statement, the `BEFORE` or `AFTER` option is specified just before the triggering statement.

In general, you use `BEFORE` or `AFTER` triggers to achieve the following results:

- Use a `BEFORE` row trigger to modify the row before the row data is written to disk. See Example 6–2 for an example of a `BEFORE` trigger.
- Use an `AFTER` row trigger to obtain and perform operations using the row ID. See Example 6–1 on page 6-11 for an example of an `AFTER` trigger.

Note: `BEFORE` row triggers are slightly more efficient than `AFTER` row triggers. With `AFTER` row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement. Alternatively, with `BEFORE` row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

If an `UPDATE` or `DELETE` statement detects a conflict with a concurrent `UPDATE` statement, then Oracle Database XE performs a transparent `ROLLBACK` and restarts the update operation. This can occur many times before the statement completes successfully. Each time the statement is restarted, the `BEFORE` statement trigger is fired again. The rollback does not undo changes to any package variables referenced in the trigger. Your package should include a counter variable to detect this situation.

Firing Triggers With the FOR EACH ROW Option

The `FOR EACH ROW` option determines whether the trigger is a row trigger or a statement trigger. If you specify `FOR EACH ROW`, then the trigger fires once for each row of the table that is affected by the triggering statement. These triggers are referred to as row-level triggers. See the use of `FOR EACH ROW` in Example 6–1 on page 6-11 and Example 6–2 on page 6-12.

The absence of the `FOR EACH ROW` option indicates that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement. These triggers are referred to as statement-level triggers and are useful for performing validation checks for the entire statement. In Example 6–6 on page 6-14, the trigger fires only once for each update of the `employees` table.

Firing Triggers Based on Conditions (WHEN Clause)

An optional trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a `WHEN` clause.

If included, the expression in the `WHEN` clause is evaluated for each row that the trigger affects. If the expression evaluates to `TRUE` for a row, then the trigger body is fired on behalf of that row. Otherwise, if the expression evaluates to `FALSE`, the trigger body is not fired. See Example 6–2 on page 6-12 for an example of the use of the `WHEN` clause in a trigger.

The expression in a `WHEN` clause must be a SQL expression, and it cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in a `WHEN` clause. A `WHEN` clause cannot be included in the definition of a statement trigger.

Firing Triggers With the INSTEAD OF Option

Use the `INSTEAD OF` option to fire the trigger instead of executing the triggering event. Unlike other types of triggers, Oracle Database XE fires the trigger instead of executing the triggering SQL DML statement.

With an `INSTEAD OF` trigger, you can run an `UPDATE`, `INSERT`, or `DELETE` statement on a complex view that otherwise could not be updated. Also, the trigger can be used to control how updates are performed on a view. The `INSTEAD OF` trigger runs transparently in the background to perform the correct actions on the underlying tables of the view. The `INSTEAD OF` option only can only be specified for a trigger created on a view and can only be activated for each row. `INSTEAD OF` triggers are valid for DML events on views. They are not valid for DDL or database events.

See "Creating a Trigger With the INSTEAD OF Option" on page 6-12.

Accessing Column Values in Row Triggers

Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement. Two correlation names exist for every column of the table being modified. There is one for the old column value and one for the new column value. These columns in the table are identified by `:OLD.`*`colum_name`* and `:NEW.`*`column_name`*. The use of `:NEW` and `:OLD` is shown in Example 6–1 on page 6-11 and Example 6–2 on page 6-12.

Depending on the type of triggering statement, certain correlation names might not have any meaning:

- A trigger fired by an `INSERT` statement has meaningful access to new column values only. Because the row is being created by the `INSERT` operation, the old values are null.
- A trigger fired by an `UPDATE` statement has access to both old and new column values for both `BEFORE` and `AFTER` row triggers.
- A trigger fired by a `DELETE` statement has meaningful access to old (:`OLD`) column values only. Because the row no longer exists after the row is deleted, the new (:`NEW`) values are `NULL` and cannot be modified.

Old and new values are available in both `BEFORE` and `AFTER` row triggers. A `new` column value can be assigned in a `BEFORE` row trigger, but not in an `AFTER` row trigger (because the triggering statement takes effect before an `AFTER` row trigger is fired). If a `BEFORE` row trigger changes the value of `NEW.column`, then an `AFTER` row trigger fired by the same statement sees the change assigned by the `BEFORE` row trigger.

Correlation names can also be used in the Boolean expression of a WHEN clause. A colon (:) must precede the OLD and NEW qualifiers when they are used in a trigger body, but a colon is not allowed when using the qualifiers in the WHEN clause.

Detecting the DML Operation That Fired a Trigger

If more than one type of DML operation can fire a trigger, such as ON INSERT or UPDATE, the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to check which type of statement fires the trigger.

Within the code of the trigger body, you can execute blocks of code depending on the kind of DML operation that fired the trigger. For an example of INSERTING and UPDATING predicates, see Example 6–6 on page 6-14.

In an UPDATE trigger, a column name can be specified with an UPDATING conditional predicate to determine if the named column is being updated. For example, assume a trigger is defined as the following:

```
CREATE OR REPLACE TRIGGER ...
... UPDATE OF salary ON employees ...
BEGIN
... IF UPDATING ('salary') THEN ... END IF;
...
```

The code in the THEN clause runs only if the triggering UPDATE statement updates the salary column. This way, the trigger can minimize its overhead when the column of interest is not being changed.

Enabled and Disabled Trigger Modes

This section discusses enabled and disabled triggers. A trigger can be in an enabled or disabled mode:

- An enabled trigger executes its trigger body if a triggering statement is entered and the trigger restriction (if any) evaluates to TRUE.
- A disabled trigger does not execute its trigger body, even if a triggering statement is entered and the trigger restriction (if any) evaluates to TRUE.

Disable a trigger if you do not want the trigger to execute, for example during maintenance activities on the database.

See "Enabling Triggers" on page 6-17 and "Disabling Triggers" on page 6-16.

Error Conditions and Exceptions in the Trigger Body

If a predefined or user-defined error condition or exception occurs during the execution of a trigger body, then all effects of the trigger body, as well as the triggering statement, are rolled back unless the error is trapped by an exception handler. Therefore, a trigger body can prevent the execution of the triggering statement by raising an exception. User-defined exceptions are commonly used in triggers that enforce complex security authorizations or integrity constraints. See "Creating a Trigger With an Exception Handler" on page 6-13 and "Handling PL/SQL Errors" on page 4-28.

Designing Triggers

This section discusses the design of triggers.

This section contains the following topics:

- Guidelines For Triggers on page 6-6
- Restrictions For Creating Triggers on page 6-6
- Privileges Needed to Work with Triggers on page 6-7

Guidelines For Triggers

Use the following guidelines when designing triggers:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Do not define triggers that duplicate features already built into Oracle Database XE. For example, do not define triggers to reject bad data if you can do the same checking through declarative integrity constraints.
- Limit the size of triggers. If the logic for a trigger requires more than 60 lines of PL/SQL code, it is better to include most of the code in a stored procedure, and call the procedure from the trigger. The size of the trigger cannot be more than 32K.
- Use triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or database application issues the statement.
- Do not create recursive triggers. For example, creating an `AFTER UPDATE` statement trigger on the `employees` table that will then issue an `UPDATE` statement on the same `employees` table, will cause the trigger to fire recursively until it has run out of memory.
- Use triggers on the database judiciously. They are executed for every user, every time the event occurs on which the trigger is created.

Restrictions For Creating Triggers

When creating triggers with PL/SQL code, there are some restrictions that are not required for standard PL/SQL blocks. The following sections discuss these restrictions.

SQL Statements Allowed in Trigger Bodies

The body of a trigger can contain DML SQL statements. It can also contain `SELECT` statements, but they must be `SELECT`... `INTO`... statements or the `SELECT` statement in the definition of a cursor.

DDL statements are not allowed in the body of a trigger and transaction control statements are not allowed in a trigger. `ROLLBACK`, `COMMIT`, and `SAVEPOINT` statements cannot be used. For system triggers, `CREATE`, `ALTER`, and `DROP TABLE` statements and `ALTER`...`COMPILE` statements are allowed.

Note: A procedure called by a trigger cannot run the previous transaction control statements because the procedure runs within the context of the trigger body.

Statements inside of a trigger can reference remote schema objects. However, pay special attention when calling remote procedures from within a local trigger. If a timestamp or signature mismatch is found during execution of the trigger, then the remote procedure is not run, and the trigger is invalidated.

System Trigger Restrictions

Only committed triggers are fired. For example, if you create a trigger that should be fired after all `CREATE` events, then the trigger itself does not fire after the creation, because the correct information about this trigger was not committed at the time when the trigger on `CREATE` events was fired.

For example, if you execute the following SQL statement, trigger `my_trigger` is not fired after the creation of `my_trigger`. Oracle Database XE does not fire a trigger that is not committed.

```
CREATE OR REPLACE TRIGGER my_trigger
  AFTER CREATE ON DATABASE
BEGIN
  NULL;
END;
```

Privileges Needed to Work with Triggers

To create a trigger in your schema, you must have the `CREATE TRIGGER` system privilege, and one of the following:

- Own the table specified in the triggering statement
- Have the `ALTER` privilege for the table in the triggering statement
- Have the `ALTER ANY TABLE` system privilege

The `CREATE TRIGGER` system privilege is included in predefined `RESOURCE` role that has been granted to the user `HR`. See "Logging in to the Database Home Page" on page 1-4.

To create a trigger on a database, you must have the `ADMINISTER DATABASE TRIGGER` privilege. If this privilege is later revoked, then you can drop the trigger, but not alter it.

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger owner, not the privilege domain of the user issuing the triggering statement. This is similar to the privilege model for stored procedures.

Managing Triggers in the Database

Triggers are another type of database object that you can manage with Object Browser. You can also create and update triggers with the SQL Commands page or SQL Editor page.

In addition, you can use SQL Command Line (SQL*Plus) to create and update triggers. For information about using SQL Command Line, see Appendix A, "Using SQL Command Line".

- Creating a Trigger With the SQL Commands Page on page 6-8
- Creating a Trigger With the Object Browser Page on page 6-9
- Viewing a Trigger With Object Browser on page 6-10
- Creating a Trigger With the AFTER and FOR EACH ROW Option on page 6-11
- Creating a Trigger With the BEFORE Option and WHEN Clause on page 6-12
- Creating a Trigger With the INSTEAD OF Option on page 6-12

- Creating a Trigger With an Exception Handler on page 6-13
- Creating a Trigger That Fires Once For Each Update on page 6-14
- Creating LOGON and LOGOFF Triggers on page 6-15
- Modifying Triggers on page 6-16
- Dropping Triggers on page 6-16
- Disabling Triggers on page 6-16
- Enabling Triggers on page 6-17
- Compiling Triggers on page 6-17

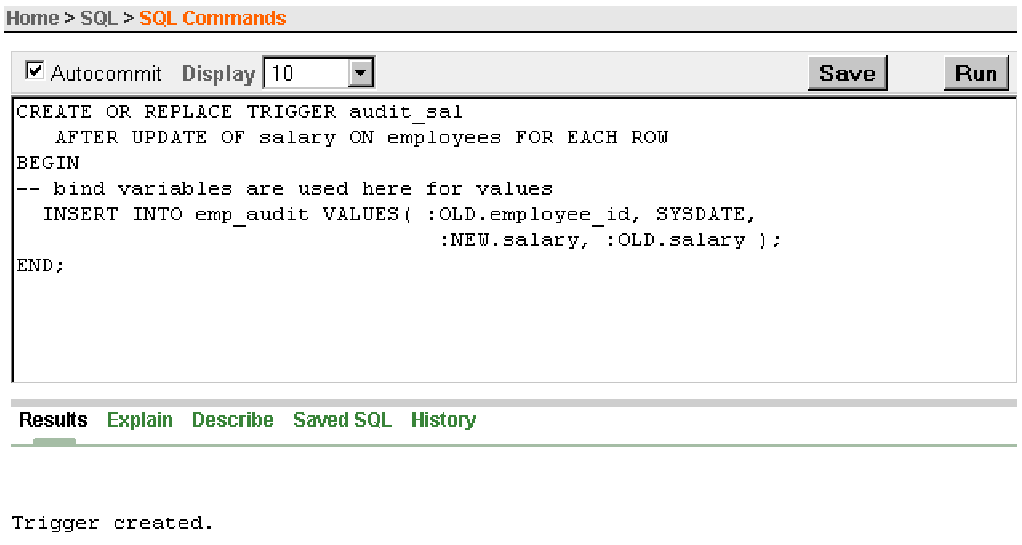
See Also: *Oracle Database Application Developer's Guide - Fundamentals* for information about the uses for and creation of triggers

Creating a Trigger With the SQL Commands Page

With the SQL Commands page, you can create and update triggers.

To create a trigger with the SQL Commands page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, you should log in as user `HR` with your password for the `HR` account.
2. On the home page, click the **SQL** icon to display the SQL page.
3. Click the **SQL Commands** icon to display the SQL Command page.
4. On the SQL Commands page, first enter the SQL statements to create any objects that are needed in the trigger body. For example, the `emp_audit` table needs to be created before creating the `audit_sal` trigger in Example 6–1 on page 6-11. If a database object is referred to in the trigger code, then that object must exist for the trigger to be valid.
5. Click the **Run** button to execute the SQL statements to create any supporting objects for the trigger. If the statements run successfully, delete the statements from the SQL Commands page. Otherwise, update the statements so they run successfully.
6. On the SQL Commands page, enter the PL/SQL code to create the trigger after any objects that are needed by the trigger are created. For an example of code to create a trigger, see Example 6–1 on page 6-11.


Home > SQL > SQL Commands
Autocommit Display 10
Save
Run
Results Explain Describe Saved SQL History

```
CREATE OR REPLACE TRIGGER audit_sal
   AFTER UPDATE OF salary ON employees FOR EACH ROW
BEGIN
-- bind variables are used here for values
  INSERT INTO emp_audit VALUES( :OLD.employee_id, SYSDATE,
                                 :NEW.salary, :OLD.salary );
END;
```

```
Trigger created.
```

7. Click the **Run** button to execute the PL/SQL code to create the trigger. Correct the code if it does not execute successfully.
8. If you want to save the PL/SQL code for future use, click the **Save** button.
9. In the **Name** field, enter a name for the saved PL/SQL code. You can also enter an optional description. Click the **Save** button to complete the action.
10. To access the saved PL/SQL code, click the **Saved SQL** tab and select the name of the saved PL/SQL code that you want to access.

See Also: *Oracle Database Express Edition Application Express User's Guide* for detailed information about using SQL Commands

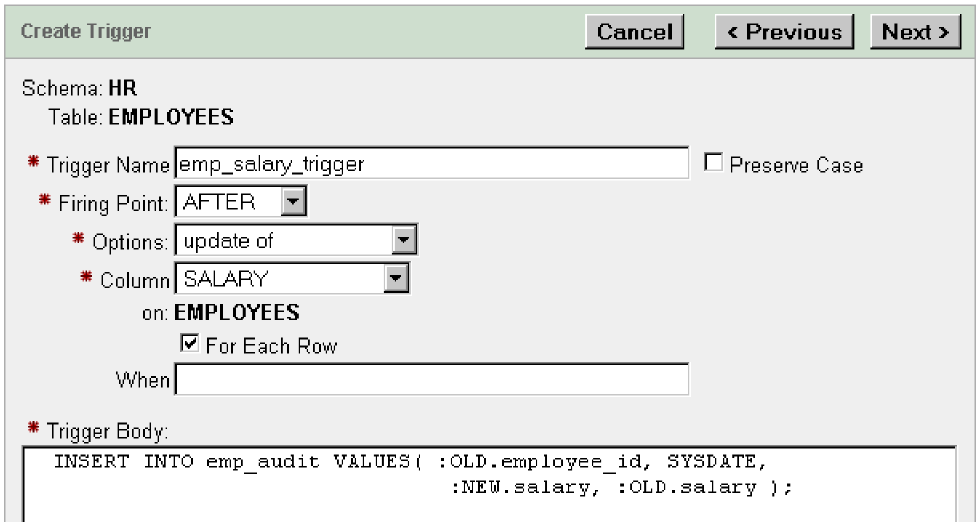
Creating a Trigger With the Object Browser Page

You can create and update triggers in the database with Object Browser.

To create a trigger with the Object Browser page:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4. To run the examples in this guide, log in as user `HR` with your password for the `HR` account.
2. Click the **Object Browser** icon on the Database Home Page.
3. Click the **Create** button, and select **Trigger** from the list.
4. Enter the name of the table (`employees`) that the trigger activity is based on and click the **Next** button. You can also select a table name from list.
5. In the **Trigger Name** field, enter the trigger name (`emp_salary_trigger`). The the **Preserve Case** box should be unchecked.
6. From the **Firing Point** list, select the firing point (`AFTER`).
7. From the **Options** list, select an option (`update of`).
8. From the **Column** list, select a column (salary).
9. Check the **For Each Row** option. Do not enter anything in the **When** field.

10. In the **Trigger Body** field, enter the code for the trigger body. See Example 6–1 on page 6-11. Note that if a database object is referred to in the trigger body code, then that object must exist for the trigger to be valid.



Schema: **HR**
Table: **EMPLOYEES**

* Trigger Name	emp_salary_trigger ☐ Preserve Case
* Firing Point:	AFTER
* Options:	update of
* Column	SALARY
on:	**EMPLOYEES**
	☑ For Each Row
When	

* Trigger Body:

```
  INSERT INTO emp_audit VALUES( :OLD.employee_id, SYSDATE,
                               :NEW.salary, :OLD.salary );
```

11. Click the **Next** button.
12. Click the **SQL** button to view the SQL statements for creating the trigger.
13. Click the **Finish** button to complete the action.

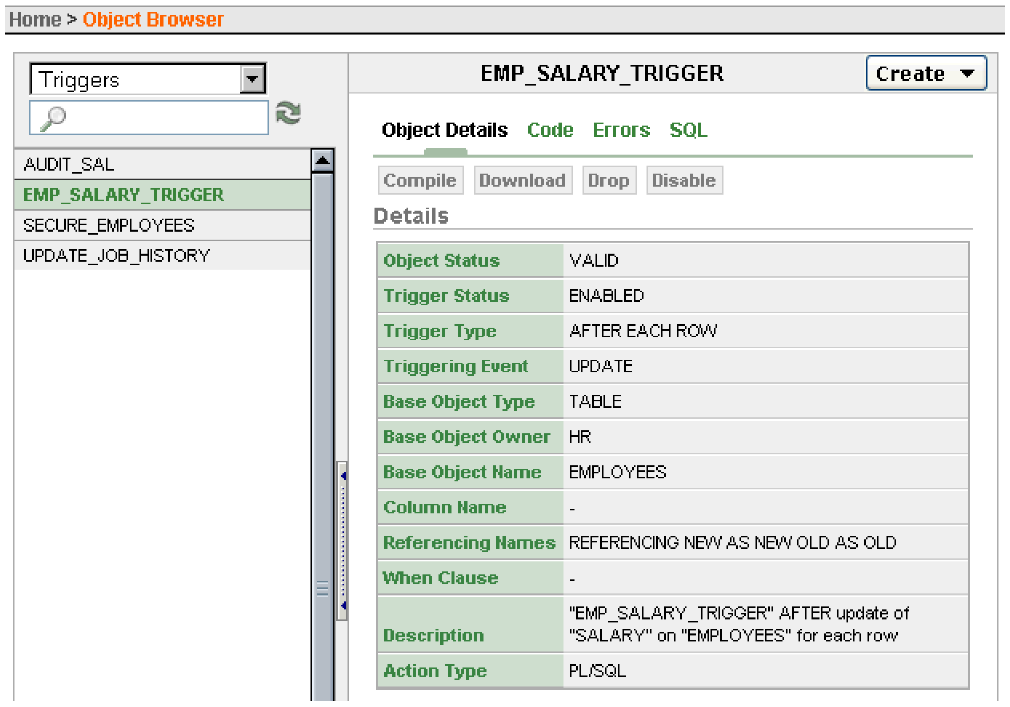
> **See Also:** *Oracle Database Express Edition Application Express User's Guide* for information about managing triggers with the Object Browser page

Viewing a Trigger With Object Browser

To find out which triggers exist in your database and display information about a specific trigger, use the Object Browser.

To display information about a trigger with Object Browser:

1. Log in to the Database Home Page. See "Logging in to the Database Home Page" on page 1-4.
2. Click the **Object Browser** icon on the Database Home Page.
3. Select **Triggers** in the object list, then select the trigger (`emp_salary_trigger`) you want to display.
4. Click the **Object Details** tab to display details about the trigger.



Object Status	VALID
Trigger Status	ENABLED
Trigger Type	AFTER EACH ROW
Triggering Event	UPDATE
Base Object Type	TABLE
Base Object Owner	HR
Base Object Name	EMPLOYEES
Column Name	-
Referencing Names	REFERENCING NEW AS NEW OLD AS OLD
When Clause	-
Description	"EMP_SALARY_TRIGGER" AFTER update of "SALARY" on "EMPLOYEES" for each row
Action Type	PL/SQL

5. Click the **Code**, **Errors**, or **SQL** tab to display additional information about the trigger.

Note that the `HR.update_job_history` trigger is fired whenever an update is performed on the `department_id` or `job_id` column of an employee record. This trigger writes a record to the `job_history` table and can raise an error if more than one update occurs in a single day.

Creating a Trigger With the AFTER and FOR EACH ROW Option

Example 6–1 shows the code for a trigger on the `employees` table. In the example, the table-level trigger fires after salaries in the `employees` table are updated and writes a record in an audit table.

With the `FOR EACH ROW` option, the trigger writes a record to the `emp_audit` table for each update. This record contains the employee ID, the date of the update, the updated salary, and the original salary. Note the use of the `:OLD.`*`column_name`* and `:NEW.`*`column_name`* to access the values in the columns before and after the update.

With the `AFTER` keyword, the trigger can also query or change the same table. Triggers can only do that after the initial changes are applied, and the table is back in a consistent state.

Because the trigger uses the `FOR EACH ROW` clause, it might be executed multiple times, such as when updating or deleting multiple rows. You might omit this clause if you just want to record the fact that the operation occurred, but not examine the data for each row.

Example 6–1 Creating a Database Trigger WIth the AFTER Option

```
-- create a table to use for with the trigger in this example if
-- it has not already been created previously
-- if the table does not exist, the trigger will be invalid
CREATE TABLE emp_audit ( emp_audit_id NUMBER(6), up_date DATE,
                         new_sal NUMBER(8,2), old_sal NUMBER(8,2) );
```

```
-- create or replace the trigger
CREATE OR REPLACE TRIGGER audit_sal
   AFTER UPDATE OF salary ON employees FOR EACH ROW
BEGIN
-- bind variables are used here for values
  INSERT INTO emp_audit VALUES( :OLD.employee_id, SYSDATE,
                               :NEW.salary, :OLD.salary );
END;
/
-- fire the trigger with an update of salary
UPDATE employees SET salary = salary * 1.01 WHERE manager_id = 122;

-- check the audit table to see if trigger was fired
SELECT * FROM emp_audit;
```

Creating a Trigger With the BEFORE Option and WHEN Clause

In Example 6–2, you define a BEFORE trigger that is fired for each row that is updated. If there are five employees in department 20, and the salaries for all the employees in the department are updated, then the trigger fires five times when those rows are updated. Note the use of the WHEN clause to restrict the firing of the trigger.

Example 6–2 Creating a Database Trigger With the BEFORE Option

```
-- create a temporary table
CREATE TABLE emp_sal_log (emp_id NUMBER, log_date DATE,
             new_salary NUMBER, action VARCHAR2(50));

CREATE OR REPLACE TRIGGER log_salary_increase -- create a trigger
  BEFORE UPDATE of salary ON employees FOR EACH ROW
  WHEN (OLD.salary < 8000)
BEGIN
  INSERT INTO emp_sal_log (emp_id, log_date, new_salary, action)
    VALUES (:NEW.employee_id, SYSDATE, :NEW.salary, 'New Salary');
END;
/
-- update the salary with the following UPDATE statement
-- trigger fires for each row that is udpated
UPDATE employees SET salary = salary * 1.01 WHERE department_id = 60;

-- view the log table
SELECT * FROM emp_sal_log;
```

Creating a Trigger With the INSTEAD OF Option

In Example 6–3 a view is created with multiple underlying tables. Note that the view in the example uses the JOIN syntax to display data from multiple tables. See "Displaying Data From Multiple Tables" on page 3-10.

Example 6–3 Creating a View That is Updated With an INSTEAD OF Trigger

```
CREATE OR REPLACE VIEW my_mgr_view AS
 SELECT ( d.department_id || ' ' ||  d.department_name) "Department",
   d.manager_id, e.first_name, e.last_name, e.email, e.hire_date "Hired On",
   e.phone_number, e.salary, e.commission_pct,
   (e.job_id || ' ' || j.job_title) "Job Class"
   FROM departments d
   JOIN employees e ON d.manager_id = e.employee_id
   JOIN jobs j ON e.job_id = j.job_id
   ORDER BY d.department_id;
```

You cannot update the employee details (first_name, last_name, email, phone_number, salary, or commission_pct) in the view in Example 6–3 with a SQL UPDATE statement. For example, the employees table cannot be updated with an UPDATE statement such as:

```
UPDATE my_mgr_view SET first_name = 'Denis'
  WHERE manager_id = 114;
-- using WHERE employee_id = 114 does not work also
```

In Example 6–4 an INSTEAD OF trigger is created that updates the underlying employees table of the view when an UPDATE statement is run on the view.

Example 6–4   Creating an INSTEAD OF Trigger for Updating a View

```
CREATE OR REPLACE TRIGGER update_my_mgr_view
  INSTEAD OF UPDATE ON my_mgr_view
  FOR EACH ROW
BEGIN
-- allow the following updates to the underlying employees table
  UPDATE employees SET
    last_name = :NEW.last_name,
    first_name = :NEW.first_name,
    email = :NEW.email,
    phone_number = :NEW.phone_number,
    salary = :NEW.salary,
    commission_pct = :NEW.commission_pct
    WHERE employee_id = :OLD.manager_id;
 END;
/
```

When the trigger in Example 6–4 is created, the following UPDATE statement can be run on the view and the INSTEAD OF trigger performs the update.

```
UPDATE my_mgr_view SET first_name = 'Denis' WHERE manager_id = 114;
```

Creating a Trigger With an Exception Handler

Example 6–5 shows how to include an exception handler with a trigger. In this example, an exception is raised if an UPDATE operation changes the manager ID of an employee.

Example 6–5   Creating a Database Trigger With an Exception Handler

```
-- create a temporary table
CREATE TABLE emp_except_log (emp_id NUMBER, mgr_id_new NUMBER,
           mgr_id_old NUMBER, log_date DATE, action VARCHAR2(50));

CREATE OR REPLACE TRIGGER emp_log_update -- create a trigger
  BEFORE UPDATE ON employees FOR EACH ROW
DECLARE
  mgrid_exception EXCEPTION;
BEGIN
  IF (:NEW.manager_id <> :OLD.manager_id) THEN
      RAISE mgrid_exception;
  END IF;
  INSERT INTO emp_except_log (emp_id, mgr_id_new, mgr_id_old, log_date, action)
    VALUES (:NEW.employee_id, :NEW.manager_id, :OLD.manager_id,
            SYSDATE, 'Employee updated');
```

```
EXCEPTION
  WHEN mgrid_exception THEN
    INSERT INTO emp_except_log (emp_id, mgr_id_new, mgr_id_old, log_date, action)
    VALUES (:NEW.employee_id, :NEW.manager_id, :OLD.manager_id,
             SYSDATE, 'Employee manager ID updated!');
END;
/
-- update employees with the following UPDATE statements, firing trigger
UPDATE employees SET salary = salary * 1.01 WHERE employee_id = 105;
-- the trigger raises an exception with this UPDATE
UPDATE employees SET manager_id = 102 WHERE employee_id = 105;

-- view the log table
SELECT * FROM emp_except_log;
```

Creating a Trigger That Fires Once For Each Update

In Example 6–6, the `FOR EACH ROW` clause is omitted so the trigger fires only once for each update of or insert into the `employees` table. Because there are two operations that fire the trigger, this example includes `IF-THEN` statements to log the specific operation that fired the trigger. The check for the `INSERTING` condition evaluates to `TRUE` only if the statement that fired the trigger is an `INSERT` statement. The check for the `UPDATING` condition evaluates to `TRUE` only if the statement that fired the trigger is an `UPDATE` statement.

Example 6–6 Creating a Trigger That Fires Only Once

```
-- create a log table
CREATE TABLE emp_update_log (log_date DATE, action VARCHAR2(50));

-- create a trigger
CREATE OR REPLACE TRIGGER log_emp_update
  AFTER UPDATE OR INSERT ON employees
DECLARE
  v_action VARCHAR2(50);
BEGIN
  IF UPDATING THEN
    v_action := 'A row has been updated in the employees table';
  END IF;
  IF INSERTING THEN
    v_action := 'A row has been inserted in the employees table';
  END IF;
  INSERT INTO emp_update_log (log_date, action)
    VALUES (SYSDATE, v_action);
END;
/

-- fire the trigger with an update
UPDATE employees SET salary = salary * 1.01 WHERE department_id = 60;
INSERT INTO employees VALUES(14, 'Belden', 'Enrique', 'EBELDEN','555.111.2222',
   '31-AUG-05', 'AC_MGR', 9000, .1, 101, 110);

-- view the log table
SELECT * FROM emp_update_log;
-- clean up: remove the inserted record
DELETE FROM employees WHERE employee_id = 14;
```

Creating LOGON and LOGOFF Triggers

You can create a trigger that performs an action when a user logs on or off the database.

In Example 6–7, a trigger is created to write a record to a log table whenever a user logs on to the `HR` account. In this example, the user name (`USER`), the type of activity (`LOGON` or `LOGOFF`), current system date (`SYSDATE`), and the number of employees in the `employees` table are written to a table. Both `SYSDATE` and `USER` are pseudocolumns that return values. See "Using ROWNUM, SYSDATE, and USER Pseudocolumns With SQL" on page 3-13.

Example 6–7 Creating a LOGON Trigger

```
-- create a table to hold the data on user logons and logoffs
CREATE TABLE hr_log_table ( user_name VARCHAR2(30), activity VARCHAR2(20),
                            logon_date DATE, employee_count NUMBER );

-- create a trigger that inserts a record in hr_log_table
-- every time a user logs on to the HR schema
CREATE OR REPLACE TRIGGER on_hr_logon
  AFTER LOGON
  ON HR.schema
DECLARE
  emp_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO emp_count FROM employees; -- count the number of employees
  INSERT INTO hr_log_table VALUES(USER, 'Log on', SYSDATE, emp_count);
END;
/
```

In Example 6–8, a trigger is created to write a record to a table whenever a user logs off the `HR` account.

Example 6–8 Creating a LOGOFF Trigger

```
-- create a trigger that inserts a record in hr_log_table
-- every time a user logs off the HR schema
CREATE OR REPLACE TRIGGER on_hr_logoff
  BEFORE LOGOFF
  ON HR.schema
DECLARE
  emp_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO emp_count FROM employees; -- count the number of employees
  INSERT INTO hr_log_table VALUES(USER, 'Log off', SYSDATE, emp_count);
END;
/
```

After you log on and log off of the `HR` account, you can check the `hr_log_table` to view results of the triggers. For example:

```
DISCONNECT
CONNECT hr/hr
SELECT * FROM hr_log_table;
```

Modifying Triggers

Similar to a stored procedure, a trigger cannot be explicitly altered. It must be replaced with a new definition. The `ALTER TRIGGER` statement is used only to recompile, enable, or disable a trigger.

When replacing a trigger, you must include the `OR REPLACE` option in the `CREATE TRIGGER` statement. The `OR REPLACE` option is provided to allow a new version of an existing trigger to replace the older version, without affecting any grants made for the original version of the trigger.

Alternatively, the trigger can be dropped using the `DROP TRIGGER` statement, and you can rerun the `CREATE TRIGGER` statement.

To drop a trigger, the trigger must be in your schema, or you must have the `DROP ANY TRIGGER` system privilege.

Dropping Triggers

When you no longer need a trigger, you can drop the trigger with Object Browser or with the SQL `DROP` command. After dropping a trigger, you can drop any dependent objects that are no longer needed.

You can disable, rather than drop, a trigger if you temporarily want to stop it from firing. See Disabling Triggers on page 6-16.

Example 6–9 shows how to drop triggers and tables used by the triggers.

Example 6–9 Dropping Triggers

```
-- first, drop the audit_sal trigger
DROP TRIGGER audit_sal;
-- then drop the table used by the trigger
DROP TABLE emp_audit;

-- drop the log_salary_increase trigger, then the table used by the trigger
DROP TRIGGER log_salary_increase;
DROP TABLE emp_sal_log;

-- drop the emp_log_update trigger, then the table used by the trigger
DROP TRIGGER emp_log_update;
DROP TABLE emp_except_log;

-- drop on_hr_logoff and on_hr_logon triggers, then drop hr_log_table
DROP TRIGGER on_hr_logon;
DROP TRIGGER on_hr_logoff;
DROP TABLE hr_log_table;
```

Disabling Triggers

You can temporarily disable a trigger. You might want to do this if:

- An object it references is not available.
- You need to perform a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

By default, triggers are enabled when first created. Disable a specific trigger using the `ALTER TRIGGER` statement with the `DISABLE` option as shown in Example 6–10.

Example 6–10 Disabling a Specific Trigger

```
ALTER TRIGGER log_emp_update DISABLE;
```

All triggers associated with a table can be disabled with one statement using the `ALTER TABLE` statement with the `DISABLE` clause and the `ALL TRIGGERS` option. Example 6–11 shows how to disable all triggers defined for the `departments` table.

Example 6–11 Disabling All Triggers on a Table

```
ALTER TABLE departments DISABLE ALL TRIGGERS;
```

Enabling Triggers

By default, a trigger is automatically enabled when it is created. However, it can be disabled if necessary. After you complete the task that requires the trigger to be disabled, reenable the trigger so that it fires when appropriate.

To enable a disabled trigger, use the `ALTER TRIGGER` statement with the `ENABLE` option as shown in Example 6–12.

Example 6–12 Enabling a Specific Trigger

```
ALTER TRIGGER log_emp_update ENABLE;
```

All triggers defined for a specific table can be enabled with one statement using the `ALTER TABLE` statement with the `ENABLE` clause with the `ALL TRIGGERS` option. Example 6–13 shows how to enable all triggers defined for the `departments` table.

Example 6–13 Enabling All Triggers for a Table

```
ALTER TABLE departments ENABLE ALL TRIGGERS;
```

Compiling Triggers

Triggers are similar to PL/SQL anonymous blocks with the addition of the :`NEW` and :`OLD` capabilities, but their compilation is different. A PL/SQL anonymous block is compiled each time it is loaded into memory. Triggers, in contrast, are fully compiled when the `CREATE TRIGGER` statement is entered, and the code is stored in the data dictionary. This means that a trigger is executed directly.

This section contains the following topics:

- Trigger Errors on page 6-17
- Dependencies for Triggers on page 6-18
- Recompiling Triggers on page 6-18

Trigger Errors

If errors occur during the compilation of a trigger, then the trigger is still created. If a DML statement fires this trigger, then the DML statement fails. You can use the `SHOW ERRORS` statement in SQL Command Line to display any compilation errors when you create a trigger in SQL, or you can use the `SELECT` statement to display the errors from the `USER_ERRORS` view as follows:

```
SELECT * FROM USER_ERRORS WHERE TYPE = 'TRIGGER';
```

Dependencies for Triggers

Compiled triggers have dependencies on database objects and become invalid if these objects, such as a table accessed from or a stored procedure called from the trigger body, is modified. Triggers that are invalidated for dependency reasons are recompiled the next time they are invoked.

You can examine the `ALL_DEPENDENCIES` view to see the dependencies for a trigger. Example 6–14 shows the use of the SQL `SELECT` statement to display the dependencies for a trigger in the `HR` schema.

Example 6–14 Viewing the Dependencies for a Trigger

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
    FROM ALL_DEPENDENCIES
    WHERE OWNER = 'HR' and TYPE = 'TRIGGER' AND NAME = 'LOG_EMP_UPDATE';
```

You can also view information about a trigger with Object Browser. See "Viewing a Trigger With Object Browser" on page 6-10.

Triggers can depend on other functions, procedures, or packages. If the function, procedure, or package specified in the trigger is dropped, then the trigger is marked invalid. An attempt is made to validate the trigger when the event occurs. If the trigger cannot be validated successfully, then it is marked `VALID WITH ERRORS`, and the event fails. For information about viewing invalid triggers in a database, see "Viewing Information With Object Reports" on page 2-6.

Note: There is an exception for `STARTUP` events. `STARTUP` events succeed even if the trigger fails. There are also exceptions for `SHUTDOWN` events and for `LOGON` events if you login as `SYSTEM`.

Recompiling Triggers

Use the `ALTER TRIGGER` statement to recompile a trigger manually. Example 6–15 shows the use of the SQL `ALTER TRIGGER` statement to recompile the `emp_log_update` trigger.

Example 6–15 Recompiling a Trigger

```
ALTER TRIGGER log_emp_update COMPILE;
-- cleanup: drop the log_emp_update trigger and emp_update_log table
DROP TRIGGER log_emp_update;
DROP TABLE emp_update_log;
```

To recompile a trigger, you must own the trigger or have the `ALTER ANY TRIGGER` system privilege.

7

Working in a Global Environment

This section discusses how to develop applications in a globalization support environment, providing information about SQL and PL/SQL Unicode programming in a global environment.

This section contains the following topics:

See Also:

- *Oracle Database Express Edition Installation Guide for Linux* or *Oracle Database Express Edition Installation Guide for Microsoft Windows* for information about setting globalization parameters and environmental variables
- *Oracle Database Globalization Support Guide* for a complete discussion of globalization support with Oracle Database Express Edition, including setting up the globalization support environment
- *Oracle Database SQL Reference* for information about date and time formats

Overview of Globalization Support

Oracle Database Express Edition globalization support enables you to store, process, and retrieve data in native languages. It ensures that database utilities, error messages, and sort order, plus date, time, monetary, numeric, and calendar conventions, automatically adapt to any native language and locale.

Oracle Database XE globalization support includes National Language Support (NLS) features. National Language Support is the ability to choose a national language and store data in a specific character set. Globalization support enables you to develop multilingual applications and software products that can be accessed and run from anywhere in the world simultaneously. An application can render content of the user interface and process data in the native language and locale preferences of the user.

This section contains the following topics:

Note: There are two distributions of Oracle Database Express Edition: one for Western Europe and the other for all languages.

- The Western European version includes a database created using a single-byte LATIN1 (WE8MSWIN1252) character set. The database can store Western European language text, such as French, Spanish, Portuguese, Italian, Dutch, German, Danish, Swedish, Norwegian, Finnish, Icelandic, as well as English. Database error messages are available in Brazilian Portuguese, English, French, German, Italian, and Spanish. The Oracle Database XE browser-based user interface is available in English only.
- The Universal version includes a multi-byte Unicode (AL32UTF8) database. The database is suitable for data of all languages, including Greek, Russian, Polish, Romanian, Hungarian, Arabic, Hebrew, Turkish, Chinese, Japanese, Korean, and all the Western European languages listed in the previous package. Both the database error messages and Oracle Database XE browser-based user interface are available in Brazilian Portuguese, Chinese (Simplified and Traditional), English, French, German, Italian, Japanese, Korean and Spanish.

The smaller, Western European version is suitable for Western European language deployment, in environments where working with an English-only development interface is acceptable. The Universal package offers support for development and deployment in all languages, and it should be used when a Unicode database is desired.

Globalization Support Features

Oracle Database XE standard features include:

- Language support

  This feature enables you to store, process, and retrieve data in native languages. Through the use of Unicode databases and datatypes, Oracle Database XE supports most contemporary languages.

  See "Setting NLS Parameters" on page 7-4.
- Territory support

  This feature supports cultural conventions that are specific to geographical locations. The default local time format, date format, numeric conventions, and monetary conventions depend on the local territory setting.

  See "Language and Territory Parameters" on page 7-5.
- Date and time formats

  This feature supports local formats for displaying the hour, day, month, and year. Time zones and daylight saving support are also available.

  See "Date and Time Parameters" on page 7-8.
- Monetary and numeric formats

  This feature supports local formats for representing currency, credit, debit symbols, and numbers.

See "Monetary Parameters" on page 7-14 and "Numeric and List Parameters" on page 7-12.

- Calendars feature

  This feature supports seven different calendar systems in use around the world: Gregorian, Japanese Imperial, ROC Official (Republic of China), Thai Buddha, Persian, English Hijrah, and Arabic Hijrah.

  See "Calendar Definitions" on page 7-11.

- Linguistic sorting

  This feature supports linguistic definitions for culturally accurate sorting and case conversion.

  See "Linguistic Sorting and Searching" on page 7-15.

- Character set support

  This feature supports a large number of single-byte, multi-byte, and fixed-width encoding schemes that are based on national, international, and vendor-specific standards.

  See *Oracle Database Express Edition Installation Guide for Linux* or *Oracle Database Express Edition Installation Guide for Microsoft Windows* for a listing of the character sets supported by Oracle Database XE.

- Character semantics

  This feature supports character semantics. It is useful for defining the storage requirements for multi-byte strings of varying widths in terms of characters instead of bytes.

  See "Length Semantics" on page 7-18.

- Unicode support

  This features supports Unicode, which is a universal encoded character set that enables you to store information in any language, using a single character set. Oracle Database Express Edition provides products such as SQL and PL/SQL for inserting and retrieving Unicode data.

  See "SQL and PL/SQL Programming with Unicode" on page 7-19.

Running the Examples

You can run the SQL examples in this chapter using the SQL Commands page, Script Editor page, or SQL Command Line (SQL*Plus). You will need to log in as the `HR` user to use the SQL statements in the examples.

For information about running SQL statements on the SQL Commands page or Script Editor page, see "Running SQL Statements" on page 3-2. For information about running SQL statements using SQL Command Line, see Appendix A, "Using SQL Command Line".

Setting Up the Globalization Support Environment

This section describes how to set up a globalization support environment.

This section contains the following topics:

- Choosing a Locale with the NLS_LANG Environment Variable on page 7-4

- Setting NLS Parameters on page 7-4
- Language and Territory Parameters on page 7-5
- Date and Time Parameters on page 7-8
- Calendar Definitions on page 7-11
- Numeric and List Parameters on page 7-12
- Monetary Parameters on page 7-14
- Linguistic Sorting and Searching on page 7-15
- Length Semantics on page 7-18

Choosing a Locale with the NLS_LANG Environment Variable

A locale is a linguistic and cultural environment in which a system or program is running. Setting the `NLS_LANG` environment parameter is the simplest way to specify locale behavior for Oracle software. It sets the language and territory used by the client application and the database. It also sets the client character set, which is the character set for data entered or displayed by a client program.

The `NLS_LANG` parameter sets the language and territory environment used by both the server session (for example, SQL statement processing) and the client application (for example, display formatting in Oracle tools).

While the default `NLS_LANG` behavior defined during installation is appropriate for most situations, you might want to modify the NLS environment dynamically during the session. To do so, you can use the `ALTER SESSION` statement to change `NLS_LANGUAGE`, `NLS_TERRITORY`, and other NLS parameters.

Note: You cannot modify the setting for the client character set with the `ALTER SESSION` statement.

The `ALTER SESSION` statement modifies only the session environment. The local client NLS environment is not modified, unless the client explicitly retrieves the new settings and modifies its local environment.

See Also:

- *Oracle Database Express Edition Installation Guide for Linux* or *Oracle Database Express Edition Installation Guide for Microsoft Windows* for information about the `NLS_LANG` environmental variable
- *Oracle Database SQL Reference* for information about the `ALTER SESSION` statement

Setting NLS Parameters

National Language Support (NLS) parameters determine the locale-specific behavior on both the client and the server. NLS parameters can be specified several ways. In this guide, altering parameters for the user session and overriding the parameters in SQL functions are discussed. Both of these techniques are accomplished through the use of SQL statements.

You can alter the NLS parameters settings by:

- Setting NLS parameters in an `ALTER SESSION` statement to override the default values that are set for the session in the initialization parameter file, or that are set by the client with environment variables. For example:

  ```
  ALTER SESSION SET NLS_SORT = french;
  ```

  Note that the changes that you make with the `ALTER SESSION` statement apply only to the current user session and are not present the next time you log in.

  See Also:

  - "Setting NLS Parameters" in *Oracle Database Globalization Support Guide* for details on setting the NLS parameters
  - *Oracle Database SQL Reference* for more information about the `ALTER SESSION` statement
  - *Oracle Database Administrator's Guide* for information about the initialization parameter file
  - *Oracle Database Reference* for information about initialization parameters used for globalization support

- Using NLS parameters within a SQL function to override the default values that are set for the session in the initialization parameter file, set for the client with environment variables, or set for the session by the `ALTER SESSION` statement. For example:

  ```
  TO_CHAR(hiredate,'DD/MON/YYYY','nls_date_language = FRENCH')
  ```

  See Also:

  - "Setting NLS Parameters" in *Oracle Database Globalization Support Guide* for information about setting the NLS parameters
  - *Oracle Database SQL Reference* for more information about SQL functions, including the `TO_CHAR` function

Additional methods for setting the NLS parameters include the use of NLS environment variables on the client, which may be platform-dependent, to specify locale-dependent behavior for the client and also to override the default values set for the session in the initialization parameter file. For example, on a Linux system:

```
% setenv NLS_SORT FRENCH
```

See Also: *Oracle Database Express Edition Installation Guide for Linux* or *Oracle Database Express Edition Installation Guide for Microsoft Windows* for information about setting globalization parameters and environmental variables

Language and Territory Parameters

Setting different NLS parameters for local territories allows 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 defined as `AMERICA`.

This section contains information about the following parameters:

- NLS_LANGUAGE Parameter on page 7-6
- NLS_TERRITORY Parameter on page 7-7

NLS_LANGUAGE Parameter

The NLS_LANGUAGE parameter can be set to any valid language name. The default is derived from the NLS_LANG setting. NLS_LANGUAGE specifies the default conventions for the following session characteristics:

- Language for server messages
- Language for day and month names and their abbreviations (specified in the SQL functions TO_CHAR and TO_DATE)
- Symbols for equivalents of AM, PM, AD, and BC
- Default sorting sequence for character data when the ORDER BY clause is specified (The GROUP BY clause uses a binary sort order unless ORDER BY is specified)

Example 7–1 and Example 7–2 show the results from setting the NLS_LANGUAGE parameter to different values. In Example 7–1, the ALTER SESSION statement is issued to set NLS_LANGUAGE to Italian.

Example 7–1 Setting NLS_LANGUAGE=ITALIAN

```
ALTER SESSION SET NLS_LANGUAGE=Italian;

-- enter a SELECT to check the format of the output after the ALTER SESSION
SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees
  WHERE employee_id IN (111, 112, 113);
```

The output from the example should be similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
Sciarra                   30-SET-97      962.5
Urman                     07-MAR-98        975
Popp                      07-DIC-99      862.5
```

Note that the abbreviations for month names are in Italian.

In Example 7–2, the ALTER SESSION statement is issued to change the language to German.

Example 7–2 Setting NLS_LANGUAGE=GERMAN

```
ALTER SESSION SET NLS_LANGUAGE=German;

SELECT last_name, hire_date, ROUND(salary/8,2) salary FROM employees
    WHERE employee_id IN (111, 112, 113);
```

The output from the example should be similar to the following:

```
LAST_NAME                 HIRE_DATE     SALARY
------------------------- --------- ----------
Sciarra                   30-SEP-97      962.5
Urman                     07-MRZ-98        975
Popp                      07-DEZ-99      862.5
```

Note that the abbreviations for the month names are now in German.

See Also:

- *Oracle Database Express Edition Installation Guide for Linux* or *Oracle Database Express Edition Installation Guide for Microsoft Windows* for information about the supported languages in the Oracle Database Express Edition
- *Oracle Database Globalization Support Guide* for more information about supported languages

NLS_TERRITORY Parameter

The `NLS_TERRITORY` parameter can be set to any valid territory name. The default is derived from the `NLS_LANG` setting. `NLS_TERRITORY` specifies the conventions for the following default date and numeric formatting characteristics:

- Date format
- Decimal character and group separator
- Local currency symbol
- ISO currency symbol
- Dual currency symbol

The territory can be modified dynamically during the session by specifying the new `NLS_TERRITORY` value in an `ALTER SESSION` statement. For example, to change the territory to France during a session, issue the following `ALTER SESSION` statement:

```
ALTER SESSION SET NLS_TERRITORY = France;
```

Modifying the `NLS_TERRITORY` parameter resets all derived NLS session parameters to default values for the new territory. Example 7–3 and Example 7–4 show the results from different settings of `NLS_TERRITORY` and `NLS_LANGUAGE`.

Example 7–3 Setting NLS_LANGUAGE=AMERICAN, NLS_TERRITORY=AMERICA

```
-- set NLS_LANAGUAGE and NLS_TERRITORY
ALTER SESSION SET NLS_LANGUAGE = American NLS_TERRITORY = America;

-- enter the following SELECT to view the format of the output for currency
SELECT TO_CHAR(salary,'L99G999D99') salary FROM employees
     WHERE employee_id IN (100, 101, 102);
```

When `NLS_TERRITORY` is set to `AMERICA` and `NLS_LANGUAGE` is set to `AMERICAN`, the results should be similar to the following:

```
SALARY
--------------------
          $24,000.00
          $17,000.00
          $17,000.00
```

In Example 7–4, an `ALTER SESSION` statement is issued to change the territory to Germany.

Example 7–4 Setting NLS_LANGUAGE=AMERICAN and NLS_TERRITORY=GERMANY

```
-- set NLS_TERRITORY to Germany for this session
ALTER SESSION SET NLS_TERRITORY = Germany;

SELECT TO_CHAR(salary,'L99G999D99') salary FROM employees
```

```
    WHERE employee_id IN (100, 101, 102);
```

The output from the example should be similar to the following:

```
SALARY
-------------------
          €24.000,00
          €17.000,00
          €17.000,00
```

Note that the currency symbol changed from dollars ($) to euros (€). The numbers have not changed because the underlying data is the same.

See Also:

- *Oracle Database Express Edition Installation Guide for Linux* or *Oracle Database Express Edition Installation Guide for Microsoft Windows* for information about the supported territories in the Oracle Database Express Edition
- *Oracle Database Globalization Support Guide* for more information about supported territories

Date and Time Parameters

Oracle Database XE enables you to control the display of the date and time, allowing different conventions for displaying the hour, day, month, and year based on the local formats. For example, in the United Kingdom, the date is displayed using the `DD/MM/YYYY` format, while China commonly uses the `YYYY-MM-DD` format.

This section contains the following topics:

- Date Formats on page 7-8
- Time Formats on page 7-10

Date Formats

Different date formats are shown in Table 7–1.

Table 7–1 Examples of Short Date Formats

Country	Description	Example
Estonia	dd.mm.yyyy	28.02.2005
Germany	dd.mm.rr	28.02.05
China	yyyy-mm-dd	2005-02-28
UK	dd/mm/yyyy	28/02/2005
US	mm/dd/yyyy	02/28/2005

This section describes the following parameters:

- NLS_DATE_FORMAT Parameter on page 7-8
- NLS_DATE_LANGUAGE Parameter on page 7-9

NLS_DATE_FORMAT Parameter The `NLS_DATE_FORMAT` parameter defines the default date format to use with the `TO_CHAR` and `TO_DATE` functions. The `NLS_TERRITORY` parameter determines the default value of the `NLS_DATE_FORMAT` parameter. The value of `NLS_DATE_FORMAT` can be any valid date format model. For example:

```
NLS_DATE_FORMAT = "MM/DD/YYYY"
```

The Oracle default date format may not always correspond to the cultural-specific convention used in a given territory. You can use the short date and long date format in SQL, using the `'DS'` and `'DL'` format models, respectively, to obtain dates in localized formats. The examples in this section show the differences among some of the date formats.

Example 7–5 shows the use of the default, short, and long date formats.

Example 7–5 Using the Default, Short, and Long Date Formats

```
-- Use an ALTER SESSION statement to change the territory to America,
-- and the language to American
ALTER SESSION SET NLS_TERRITORY = America NLS_LANGUAGE = American;

-- After the session is altered, select the dates with the format models
SELECT hire_date, TO_CHAR(hire_date,'DS') "Short",
  TO_CHAR(hire_date,'DL') "Long" FROM employees
  WHERE employee_id IN (111, 112, 113);
```

The results of the query in Example 7–5 should be similar to the following:

```
HIRE_DATE 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
```

To add string literals to the date format, enclose the string literal with double quotes. Note that when double quotation marks are included in the date format, the entire value must be enclosed by single quotation marks. For example:

```
NLS_DATE_FORMAT = '"Date: "MM/DD/YYYY'
```

NLS_DATE_LANGUAGE Parameter The `NLS_DATE_LANGUAGE` parameter specifies the language for the day and month produced by the `TO_CHAR` and `TO_DATE` functions. `NLS_DATE_LANGUAGE` overrides the language that is specified implicitly by `NLS_LANGUAGE`. The `NLS_DATE_LANGUAGE` parameter has the same syntax as the `NLS_LANGUAGE` parameter, and all supported languages are valid values.

The `NLS_DATE_LANGUAGE` parameter also determines the language used for:

- Month and day abbreviations returned by the `TO_CHAR` and `TO_DATE` functions
- Month and day abbreviations used by the default date format (`NLS_DATE_FORMAT`)

Example 7–6 shows how to use `NLS_DATE_LANGUAGE` to set the date language to French.

Example 7–6 Setting NLS_DATE_LANGUAGE=FRENCH: Month and Day

```
-- set NLS_DATE_LANAGUAGE for this user session
ALTER SESSION SET NLS_DATE_LANGUAGE = FRENCH;

-- display the current system date
SELECT TO_CHAR(SYSDATE, 'Day:Dd Month yyyy') FROM DUAL;
```

The output from the example should be similar to the following, depending on the current system date:

```
TO_CHAR(SYSDATE,'DAY:DDMON
--------------------------
Jeudi    :06 Octobre   2005
```

The default date format uses the month abbreviations determined by the `NLS_DATE_LANGUAGE` parameter. For example, if the default date format is `DD-MON-YYYY` and `NLS_DATE_LANGUAGE = FRENCH`, then insert a date as follows:

```
INSERT INTO table_name VALUES ('12-Févr.-1997');
```

See Also: *Oracle Database SQL Reference* for information about date format models

Time Formats

Different time formats are shown in Table 7–2.

Table 7–2 Examples of Time Formats

Country	Description	Example
Estonia	hh24:mi:ss	13:50:23
Germany	hh24:mi:ss	13:50:23
China	hh24:mi:ss	13:50:23
UK	hh24:mi:ss	13:50:23
US	hh:mi:ssxff am	1:50:23.555 PM

This section describes the following parameters:

- NLS_TIMESTAMP_FORMAT Parameter on page 7-10
- NLS_TIMESTAMP_TZ_FORMAT Parameter on page 7-10

NLS_TIMESTAMP_FORMAT Parameter The `NLS_TIMESTAMP_FORMAT` parameter defines the default date format for the `TIMESTAMP` and `TIMESTAMP WITH LOCAL TIME ZONE` datatypes. The `NLS_TERRITORY` parameter determines the default value of `NLS_TIMESTAMP_FORMAT`. The value of `NLS_TIMESTAMP_FORMAT` can be any valid datetime format model.

The following example shows a value for `NLS_TIMESTAMP_FORMAT`:

```
NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH:MI:SS.FF'
```

NLS_TIMESTAMP_TZ_FORMAT Parameter The `NLS_TIMESTAMP_TZ_FORMAT` parameter defines the default date format for the `TIMESTAMP` and `TIMESTAMP WITH LOCAL TIME ZONE` datatypes. It is used with the `TO_CHAR` and `TO_TIMESTAMP_TZ` functions. The `NLS_TERRITORY` parameter determines the default value of the `NLS_TIMESTAMP_TZ_FORMAT` parameter. The value of `NLS_TIMESTAMP_TZ_FORMAT` can be any valid datetime format model.

The format value must be surrounded by quotation marks. For example:

```
NLS_TIMESTAMP_TZ_FORMAT = 'YYYY-MM-DD HH:MI:SS.FF TZH:TZM'
```

In Example 7–7 the `TO_TIMESTAMP_TZ` function uses the format value that was specified for `NLS_TIMESTAMP_TZ_FORMAT`.

Example 7–7 Setting NLS_TIMESTAMP_TZ_FORMAT

```
-- display August 20, 2005 using the format of NLS_TIMPSTAMP_TZ_FORMAT
SELECT TO_TIMESTAMP_TZ('2005-08-20, 05:00:00.55 America/Los_Angeles',
  'yyyy-mm-dd hh:mi:ss.ff TZR') "TIMESTAMP_TZ Format" FROM DUAL;
```

The output from the example should be similar to the following:

```
TIMESTAMP_TZ Format
---------------------------------------------------
20-AUG-05 05.00.00.550000000 AM AMERICA/LOS_ANGELES
```

Calendar Definitions

This section contains the following topics:

- Calendar Formats on page 7-11
- NLS_CALENDAR Parameter on page 7-12

Calendar Formats

The following calendar information is stored for each territory:

- First Day of the Week on page 7-11
- First Calendar Week of the Year on page 7-11
- Number of Days and Months in a Year on page 7-11
- First Year of Era on page 7-12

First Day of the Week Some cultures consider Sunday to be the first day of the week. Others consider Monday to be the first day of the week.

The first day of the week is determined by the `NLS_TERRITORY` parameter.

First Calendar Week of the Year Some countries use week numbers for scheduling, planning, and bookkeeping. Oracle supports this convention. In the ISO standard, the week number can be different from the week number of the calendar year. For example, `1st Jan 1988` is in ISO week number 53 of 1987. An ISO week starts on Monday and ends on Sunday.

To support the ISO standard, Oracle provides the IW date format element. It returns the ISO week number.

The first calendar week of the year is determined by the `NLS_TERRITORY` parameter.

Number of Days and Months in a Year Oracle supports six calendar systems in addition to the Gregorian calendar, which is the default. The six calendar systems are:

- Japanese Imperial—uses the same number of months and days as the Gregorian calendar, but the year starts with the beginning of each Imperial Era
- ROC Official—uses the same number of months and days as the Gregorian calendar, but the year starts with the founding of the Republic of China
- Persian—has 31 days for each of the first 6 months. The next 5 months have 30 days each. The last month has either 29 days or 30 days (leap year).
- Thai Buddha—uses a Buddhist calendar
- Arabic Hijrah—has 12 months with 354 or 355 days
- English Hijrah—has 12 months with 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.

NLS_CALENDAR Parameter

Many different calendar systems are in use throughout the world. The `NLS_CALENDAR` parameter specifies which calendar system Oracle Database XE uses. The default value is `Gregorian`. The value can be any valid calendar format name.

The `NLS_CALENDAR` parameter can have one of the following values:

- Arabic Hijrah
- English Hijrah
- Gregorian
- Japanese Imperial
- Persian
- ROC Official (Republic of China)
- Thai Buddha

In Example 7–8, the `NLS_CALENDAR` parameter is set to English Hijrah.

Example 7–8 Setting NLS_CALENDAR='English Hijrah'

```
-- set NLS_CALENDAR with ALTER SESSION
ALTER SESSION SET NLS_CALENDAR='English Hijrah';

-- display the current system date
SELECT SYSDATE FROM DUAL;
```

The output from the example should be similar to the following, depending on the current system date:

```
SYSDATE
--------------------
24 Ramadan      1422
```

Numeric and List Parameters

This section contains the following topics:

- Numeric Formats on page 7-12
- NLS_NUMERIC_CHARACTERS Parameter on page 7-13

Numeric Formats

The database must know the number-formatting convention used in each session to interpret numeric strings correctly. For example, the database needs to know whether numbers are entered with a period or a comma as the decimal character (234.00 or 234,00). Similarly, applications must be able to display numeric information in the format expected at the client site.

Examples of numeric formats are shown in Table 7–3.

Table 7–3 Examples of Numeric Formats

Country	Numeric Formats
Estonia	1 234 567,89
Germany	1.234.567,89
China	1,234,567.89
UK	1,234,567.89
US	1,234,567.89

Numeric formats are derived from the `NLS_TERRITORY` parameter setting, but they can be overridden by the `NLS_NUMERIC_CHARACTERS` parameter.

NLS_NUMERIC_CHARACTERS Parameter

The `NLS_NUMERIC_CHARACTERS` parameter specifies the group separator and decimal character. The group separator is the character that separates integer groups to show thousands and millions, for example. The group separator is the character returned by the `G` number format model. The decimal character separates the integer and decimal parts of a number. Setting the `NLS_NUMERIC_CHARACTERS` parameter overrides the default values derived from the setting of `NLS_TERRITORY`. The value can be any two valid numeric characters for the group separator and decimal character.

Any character can be the decimal character or group separator. The two characters specified must be single-byte, and the characters must be different from each other. The characters cannot be a numeric character or any of the following characters: plus sign (+), minus sign (-), less than sign (<), greater than sign (>). Either character can be a space.

To set the decimal character to a comma and the grouping separator to a period, specify the `NLS_NUMERIC_CHARACTERS` parameter as follows:

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ",.";
```

SQL statements can include numbers represented as numeric or text literals. Numeric literals are not enclosed in quotation marks. They are part of the SQL language syntax, and always use a period as the decimal character and never contain a group separator. Text literals are enclosed in single quotation marks. They are implicitly or explicitly converted to numbers, if required, according to the current NLS settings.

The `SELECT` statement in Example 7–9 formats 4000 with the decimal character and group separator specified in the `ALTER SESSION` statement:

Example 7–9 Setting NLS_NUMERIC_CHARACTERS=",."

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ",.";
SELECT TO_CHAR(4000, '9G999D99') FROM DUAL;
```

The output from the example should be similar to the following:

```
TO_CHAR(4
---------
 4.000,00
```

Monetary Parameters

Oracle Database XE enables you to define radix symbols and thousands separators by locales. For example, in the US, the decimal point is a period (.), while it is a comma (,) in France. Because $1,234 has different meanings in different countries, it is important to display the amount appropriately by locale.

This section contains the following topics:

- Currency Formats on page 7-14
- NLS_CURRENCY Parameter on page 7-14
- NLS_ISO_CURRENCY Parameter on page 7-15
- NLS_DUAL_CURRENCY Parameter on page 7-15

Currency Formats

Different currency formats are used throughout the world. Some typical formats are shown in Table 7–4.

Table 7–4 Currency Format Examples

Country	Example
Estonia	1 234,56 kr
Germany	1.234,56€
China	¥1,234.56
UK	£1,234.56
US	$1,234.56

NLS_CURRENCY Parameter

The `NLS_CURRENCY` parameter specifies the character string returned by the `L` number format model, the local currency symbol. Setting `NLS_CURRENCY` overrides the default setting defined implicitly by `NLS_TERRITORY`. The value can be any valid currency symbol string, as shown in Example 7–10.

Example 7–10 Displaying the Local Currency Symbol

```
-- select and format the salary column from employees
SELECT TO_CHAR(salary, 'L099G999D99') "salary" FROM employees
  WHERE salary > 11000;
```

The output from the example should be similar to the following:

```
SALARY
---------------------
             $024,000.00
             $017,000.00
             $017,000.00
             $012,000.00
             $014,000.00
             $013,500.00
             $012,000.00
             $011,500.00
             $013,000.00
             $012,000.00
```

NLS_ISO_CURRENCY Parameter

The NLS_ISO_CURRENCY parameter specifies the character string returned by the C number format model, the ISO currency symbol. Setting NLS_ISO_CURRENCY overrides the default value defined implicitly by NLS_TERRITORY. The value can be any valid string.

Local currency symbols can be ambiguous. For example, a dollar sign ($) can refer to U.S. dollars or Australian dollars. ISO specifications define unique currency symbols for specific territories or countries. For example, the ISO currency symbol for the U.S. dollar is USD. The ISO currency symbol for the Australian dollar is AUD.

The NLS_ISO_CURRENCY parameter has the same syntax as the NLS_TERRITORY parameter, and all supported territories are valid values.

To specify the ISO currency symbol for France, set NLS_ISO_CURRENCY as shown in Example 7–11.

Example 7–11   Setting NLS_ISO_CURRENCY=FRANCE

```
-- set NLS_ISO_CURRENCY to France
ALTER SESSION SET NLS_ISO_CURRENCY = FRANCE;

-- display the salary of selected employees
SELECT TO_CHAR(salary, 'C099G999D99') "Salary" FROM employees
  WHERE department_id = 60;
```

The output from the example should be similar to the following:

```
Salary
--------------------
         EUR009,000.00
         EUR006,000.00
         EUR004,800.00
         EUR004,800.00
         EUR004,200.00
```

NLS_DUAL_CURRENCY Parameter

Use the NLS_DUAL_CURRENCY parameter to override the default dual currency symbol defined implicitly by NLS_TERRITORY. The value can be any valid symbol.

NLS_DUAL_CURRENCY was introduced to support the euro currency symbol during the euro transition period.

Linguistic Sorting and Searching

Different languages have their own sorting rules. Some languages are collated according to the letter sequence in the alphabet, some according to the number of stroke counts in the letter, and some are ordered by the pronunciation of the words. Treatment of letter accents also differs among languages. For example, in Danish, Æ is sorted after Z, while Y and Ü are considered to be variants of the same letter.

You can define how to sort data by using linguistic sort parameters. The basic linguistic definition treats strings as sequences of independent characters.

This section contains the following topics:

- NLS_SORT Parameter on page 7-16
- NLS_COMP Parameter on page 7-17

- Case-Insensitive and Accent-Insensitive Searching on page 7-18

NLS_SORT Parameter

The NLS_SORT parameter specifies the collating (linguistic sort) sequence for ORDER BY queries. It overrides the default NLS_SORT value that is derived from the NLS_LANGUAGE parameter. The value of NLS_SORT can be BINARY or any valid linguistic sort name:

NLS_SORT = BINARY | *sort_name*

If the value is BINARY, then the collating sequence is based on the numeric code of the characters in the underlying encoding scheme. Depending on the datatype, this will either be in the binary sequence order of the database character set or the national character set. If the value is a named linguistic sort, sorting is based on the order of the defined sort. Most, but not all, languages supported by the NLS_LANGUAGE parameter also support a linguistic sort with the same name.

Spain traditionally treats ch, ll, and ñ as letters of their own, ordered after c, l, and n, respectively. Example 7–12 and Example 7–13 illustrate the effect of using a Spanish sort against the employee names Chen and Chung. In Example 7–12, the NLS_SORT parameter is set to BINARY.

In Example 7–12, the LIKE comparison operator is used to specify the records to return with the query. For information about LIKE, see "Restricting Data Using the WHERE Clause" on page 3-6.

Example 7–12 Setting NLS_SORT to BINARY

```
-- set the NLS_SORT for this user session
ALTER SESSION SET NLS_SORT=binary;

-- select the last name of those employees whose last name begin with C
SELECT last_name FROM employees
   WHERE last_name LIKE 'C%' ORDER BY last_name;
```

The output from the example should be similar to the following:

```
LAST_NAME
--------------
Cabrio
Cambrault
Cambrault
Chen
Chung
Colmenares
```

In Example 7–13, the NLS_SORT parameter is set to SPANISH_M.

Example 7–13 Setting NLS_SORT to Spanish

```
-- set the NLS_SORT for this user session
ALTER SESSION SET NLS_SORT=spanish_m;

-- select the last name of those employees whose last name begin with C
SELECT last_name FROM employees
   WHERE last_name LIKE 'C%' ORDER BY last_name;
```

The output from the example should be similar to the following:

```
LAST_NAME
```

```
--------------
Cabrio
Cambrault
Cambrault
Colmenares
Chen
Chung
```

Note that the order of last names in the output from the SELECT statement in Example 7–12 and Example 7–13 is different.

> **See Also:** *Oracle Database Globalization Support Guide* for more information about supported linguistic sorts

NLS_COMP Parameter

When using comparison operators, characters are compared according to their binary codes in the designated encoding scheme. A character is greater than another if it has a higher binary code. Because the binary sequence of characters may not match the linguistic sequence for a particular language, those comparisons might not be linguistically correct.

The value of the NLS_COMP parameter affects the comparison behavior of SQL operations. The value can be BINARY (default) or LINGUISTIC. You can use the NLS_COMP parameter to avoid the cumbersome process of using the NLSSORT function in SQL statements when you want to perform a linguistic comparison instead of a binary comparison. When NLS_COMP is set to LINGUISTIC, SQL performs a linguistic comparison based on the value of the NLS_SORT parameter.

Example 7–14 and Example 7–15 illustrate the effect of performing a binary comparison follow by a Spanish linguistic sensitive comparison against the employee names. In Example 7–14 the NLS_COMP parameter is set to BINARY while NLS_SORT is set to Spanish.

Example 7–14 Setting NLS_COMP to BINARY

```
-- set NLS_SORT and NLS_COMP for this user session
ALTER SESSION SET NLS_SORT=spanish_m NLS_COMP=binary;

-- select the last name of those employees whose last name begin with C
SELECT last_name FROM employees
  WHERE last_name LIKE 'C%';
```

The output from the example should be similar to the following:

```
LAST_NAME
--------------
Cabrio
Cambrault
Cambrault
Chen
Chung
Colmenares
```

In Example 7–15 the NLS_COMP parameter is set to LINGUISTIC while NLS_SORT is set to Spanish.

Example 7–15 Setting NLS_COMP to LINGUISTIC

```
-- set NLS_SORT and NLS_COMP for this user session
ALTER SESSION SET NLS_SORT=spanish_m NLS_COMP=linguistic;

-- select the last name of those employees whose last name begin with C
SELECT last_name FROM employees
  WHERE last_name LIKE 'C%';
```

The output from the example should be similar to the following:

```
LAST_NAME
--------------
Cabrio
Cambrault
Cambrault
Colmenares
```

Note the difference in the output from Example 7–14 and Example 7–15. In Spanish `ch` is treated as a separate character that follows `c`, so `ch` is excluded when a Spanish linguistic-sensitive comparison is performed in Example 7–15.

Case-Insensitive and Accent-Insensitive Searching

Operations inside of a database are sensitive to the case and the accents of the characters. Sometimes, you might need to perform case-insensitive or accent-insensitive comparisons. Use the `NLS_SORT` session parameter to specify a case-insensitive or accent-insensitive sort.

To specify a case-insensitive or accent-insensitive sort:

- Append `_CI` to an Oracle sort name for a case-insensitive sort. For example:

  `BINARY_CI`: accent-sensitive and case-insensitive binary sort
  `GENERIC_M_CI`: accent-sensitive and case-insensitive `GENERIC_M` sort

- Append `_AI` to an Oracle sort name for an accent-insensitive and case-insensitive sort. For example:

  `BINARY_AI`: accent-insensitive and case-insensitive binary sort
  `FRENCH_M_AI`: accent-insensitive and case-insensitive `FRENCH_M` sort

Length Semantics

In single-byte character sets, the number of bytes and the number of characters in a string are the same. In multi-byte 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 to define the storage requirements for multi-byte strings of varying widths. For example, in a Unicode database (AL32UTF8), suppose that you need to define a `VARCHAR2` column that can store up to five Chinese characters together 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.

The expressions in the following list use byte semantics. Note the `BYTE` qualifier in the `VARCHAR2` expression and the `B` suffix in the SQL function name.

- `VARCHAR2(20 BYTE)`
- `SUBSTRB(`*`string`*`, 1, 20)`

The expressions in the following list use character semantics. Note the `CHAR` qualifier in the `VARCHAR2` expression.

- `VARCHAR2(20 CHAR)`
- `SUBSTR(`*`string`*`, 1, 20)`

This section contains the following topic:

- NLS_LENGTH_SEMANTICS Parameter on page 7-19

NLS_LENGTH_SEMANTICS Parameter

The `NLS_LENGTH_SEMANTICS` parameter specifies `BYTE` (default) or `CHAR` semantics. By default, the character datatypes `CHAR` and `VARCHAR2` are specified in bytes, not characters. Therefore, the specification `CHAR(20)` in a table definition allows 20 bytes for storing character data.

The `NLS_LENGTH_SEMANTICS` parameter enables you to create `CHAR`, `VARCHAR2`, and `LONG` columns using either byte-length or character-length semantics. `NCHAR`, `NVARCHAR2`, `CLOB`, and `NCLOB` columns are always character-based. Existing columns are not affected.

Example 7–16 shows an example of creating a table. When the database character set is WE8MSWIN1252, the `last_name` column of the table can hold up to 10 Western European characters, occupying a maximum of 10 bytes. When the database character set is Unicode (AL32UTF8), the `last_name` column can still hold up to 10 Unicode characters regardless of the language; however, it can occupy a maximum of 40 bytes.

Example 7–16 Setting Length Semantics and Creating a Table

```
-- reset NLS parameters back to default here
ALTER SESSION SET NLS_LANGUAGE = American NLS_TERRITORY = America;
CREATE TABLE temp_employees_table
( employee_id NUMBER(4), last_name VARCHAR2(10 CHAR), job_id VARCHAR2(9),
  manager_id NUMBER(4), hire_date DATE, salary NUMBER(7,2),
  department_id NUMBER(2)) ;
-- cleanup: drop the table
DROP TABLE temp_employees_table;
```

See Also: "Length Semantics for Character Datatypes" in *Oracle Database Concepts*

SQL and PL/SQL Programming with Unicode

This section describes Unicode-related features in SQL and PL/SQL that you can deploy for multiple language applications. Oracle Database XE provides products such as SQL and PL/SQL for inserting and retrieving 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.

This section contains the following topics:

- Overview of Unicode on page 7-20
- SQL NCHAR Datatypes on page 7-20

- Unicode String Literals on page 7-21
- NCHAR Literal Replacement on page 7-22

See Also:

- *Oracle Database SQL Reference* for information about SQL
- *Oracle Database PL/SQL User's Guide and Reference* for information about PL/SQL

Overview of Unicode

Unicode is a universal encoded character set that enables you to store information in any language, using a single character set. Unicode provides a unique code value for every character, regardless of the platform, program, or language.

Unicode has the following advantages:

- It simplifies character set conversion and linguistic sort functions.
- It improves performance compared with native multi-byte character sets.
- It supports the Unicode datatype based on the Unicode standard.

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 `CHAR` datatypes.
- You can support multiple language data in specific columns by using Unicode datatypes. You can store Unicode characters into columns of the SQL `NCHAR` datatypes regardless of how the database character set has been defined. The `NCHAR` datatype is an exclusively Unicode datatype.

SQL NCHAR Datatypes

There are two SQL `NCHAR` datatypes:

- NCHAR Datatype on page 7-20
- NVARCHAR2 Datatype on page 7-21

NCHAR Datatype

When you define a table column or a PL/SQL variable as the `NCHAR` datatype, the length is specified as the number of characters. For example, the following statement creates a column with a maximum length of 30 characters:

```
CREATE TABLE table1 (column1 NCHAR(30));
```

The maximum number of bytes for the column is determined as follows:

maximum number of bytes =
(maximum number of characters) x (maximum number of bytes for each character)

For example, if the national character set is UTF8, then the maximum byte length is 30 characters times 3 bytes for each character, or 90 bytes.

The national character set, which is used for all `NCHAR` datatypes, is defined when the database is created. The national character set can be either UTF8 or AL16UTF16. The default is AL16UTF16.

The maximum column size allowed is 2000 characters when the national character set is UTF8 and 1000 when it is AL16UTF16. The actual data is subject to the maximum byte limit of 2000. The two size constraints must be satisfied at the same time. In PL/SQL, the maximum length of the NCHAR data is 32,767 bytes. You can define an NCHAR variable of up to 32,767 characters, but the actual data cannot exceed 32,767 bytes. If you insert a value that is shorter than the column length, then Oracle pads the value with blanks to whichever length is smaller: maximum character length or maximum byte length.

NVARCHAR2 Datatype

The NVARCHAR2 datatype specifies a variable-length character string that uses the national character set. When you create a table with an NVARCHAR2 column, you specify the maximum number of characters for the column. Lengths for NVARCHAR2 are in units of characters, just as for NCHAR. Oracle Database XE subsequently stores each value in the column exactly as you specify it, if the value does not exceed the maximum length of the column. Oracle Database XE does not pad the string value to the maximum length.

The maximum column size allowed is 4000 characters when the national character set is UTF8, and it is 2000 when AL16UTF16. The maximum length of an NVARCHAR2 column in bytes is 4000. Both the byte limit and the character limit must be met, so the maximum number of characters that is allowed in an NVARCHAR2 column is the number of characters that can be written in 4000 bytes.

In PL/SQL, the maximum length for an NVARCHAR2 variable is 32,767 bytes. You can define NVARCHAR2 variables up to 32,767 characters, but the actual data cannot exceed 32,767 bytes.

The following statement creates a table with one NVARCHAR2 column whose maximum length in characters is 2000 and maximum length in bytes is 4000.

```
CREATE TABLE table2 (column2 NVARCHAR2(2000));
```

Unicode String Literals

You can input Unicode string literals in SQL and PL/SQL as follows:

- Put the letter N before a string literal that is enclosed with single quotation marks. This explicitly indicates that the following string literal is an NCHAR string literal. For example, N'résumé' is an NCHAR string literal. See "NCHAR Literal Replacement" on page 7-22 for limitations of this method.
- Use the NCHR(*n*) SQL function. The NCHR(*n*) SQL function returns a unit of character code in the national character set, which is AL16UTF16 or UTF8. The result of concatenating several NCHR(*n*) functions is NVARCHAR2 data. In this way, you can bypass the client and server character set conversions and create an NVARCHAR2 string directly. For example, NCHR(32) represents a blank character.

  Because NCHR(*n*) is associated with the national character set, portability of the resulting value is limited to applications that use the same national character set. If this is a concern, then use the UNISTR function to remove portability limitations.
- Use the UNISTR('*string*') SQL function. The UNISTR('*string*') function converts a string to the national character set. To ensure portability and to preserve data, include only ASCII characters and Unicode encoding in the following form: \xxxx, where xxxx is the hexadecimal value of a character code value in UTF-16 encoding format. For example, UNISTR('G\0061ry') represents 'Gary'. The ASCII characters are converted to the database character set and then to the national character set. The Unicode encoding is converted directly to the national character set.

The last two methods can be used to encode any Unicode string literals.

NCHAR Literal Replacement

Being part of a SQL or PL/SQL statement, the text of any 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 in the client character set, determined by the character set defined in the `NLS_LANG` parameter. On the server side the statement is in the database character set.

When the SQL or PL/SQL statement is transferred from client to the database, its character set is converted accordingly. If the database character set does not contain all characters used in the text literals, the data is lost in this conversion. This affects `NCHAR` string literals more than the `CHAR` text literals, this is because the `N'` literals is designed to be independent of the database character set, and it should be able to include any data that the client character set allows.

To avoid data loss during conversion to an incompatible database character set, you can activate the `NCHAR` literal replacement functionality. It transparently replaces the N′ literals on the client side into an internal format, the database then decodes this to Unicode when the statement is executed. You can set the client environment variable `ORA_NCHAR_LITERAL_REPLACE` to `TRUE` to enable this functionality. By default, the functionality is switched off to maintain backward compatibility.

Locale-Dependent SQL Functions with Optional NLS Parameters

All SQL functions whose behavior depends on globalization support conventions allow NLS parameters to be specified. These functions are:

```
TO_CHAR
TO_DATE
TO_NUMBER
NLS_UPPER
NLS_LOWER
NLS_INITCAP
NLSSORT
```

Explicitly specifying the optional NLS parameters for these functions enables the functions to be evaluated independently of the session's NLS parameters. This feature can be important for SQL statements that contain numbers and dates as string literals.

For example, the following query is evaluated correctly if the language specified for dates is `AMERICAN` and the calender is specified as `GREGORIAN`.

Example 7–17 Setting NLS_DATE_LANGUAGE=American, NLS_CALENDAR=Gregorian

```
ALTER SESSION SET NLS_DATE_LANGUAGE=American;
ALTER SESSION SET NLS_CALENDAR=Gregorian;
SELECT last_name FROM employees WHERE hire_date > '01-JAN-1999';
```

The previous query can be made independent of the current date language by using a statement similar to the following:

Example 7–18 Setting NLS_LANGUAGE in a Query

```
SELECT last_name FROM employees
     WHERE hire_date > TO_DATE('01-JAN-1999','DD-MON-YYYY',
     'NLS_DATE_LANGUAGE = AMERICAN');
```

In this way, SQL statements that are independent of the session language can be defined where necessary. These statements are necessary when string literals appear in SQL statements in views, `CHECK` constraints, or triggers.

Note: Only SQL statements that must be independent of the session NLS parameter values should explicitly specify optional NLS parameters in locale-dependent SQL functions. Using session default values for NLS parameters in SQL functions usually results in better performance.

All character functions support both single-byte and multi-byte characters. Except where explicitly stated, character functions operate character by character, rather than byte by byte.

The remainder of this section contains the following topics:

- Default Values for NLS Parameters in SQL Functions on page 7-23
- Specifying NLS Parameters in SQL Functions on page 7-23
- Unacceptable NLS Parameters in SQL Functions on page 7-25

Default Values for NLS Parameters in SQL Functions

When SQL functions evaluate views and triggers, default values from the current session are used for the NLS function parameters. When SQL functions evaluate `CHECK` constraints, they use the default values that were specified for the NLS parameters when the database was created.

Specifying NLS Parameters in SQL Functions

NLS parameters are specified in SQL functions as '*`parameter`* = *`value`*'. For example:

```
'NLS_DATE_LANGUAGE = AMERICAN'
```

The following NLS parameters can be specified in SQL functions:

```
NLS_DATE_LANGUAGE
NLS_NUMERIC_CHARACTERS
NLS_CURRENCY
NLS_ISO_CURRENCY
NLS_DUAL_CURRENCY
NLS_CALENDAR
NLS_SORT
```

Table 7–5 shows which NLS parameters are valid for specific SQL functions.

Table 7–5 SQL Functions and Their Valid NLS Parameters

SQL Function	Valid NLS Parameters
`TO_DATE`	`NLS_DATE_LANGUAGE,` `NLS_CALENDAR`
`TO_NUMBER`	`NLS_NUMERIC_CHARACTERS,` `NLS_CURRENCY,` `NLS_DUAL_CURRENCY,` `NLS_ISO_CURRENCY,`

Table 7–5 (Cont.) SQL Functions and Their Valid NLS Parameters

SQL Function	Valid NLS Parameters
`TO_CHAR`	`NLS_DATE_LANGUAGE, NLS_NUMERIC_CHARACTERS, NLS_CURRENCY, NLS_ISO_CURRENCY, NLS_DUAL_CURRENCY, NLS_CALENDAR`
`TO_NCHAR`	`NLS_DATE_LANGUAGE, NLS_NUMERIC_CHARACTERS, NLS_CURRENCY, NLS_ISO_CURRENCY, NLS_DUAL_CURRENCY, NLS_CALENDAR`
`NLS_UPPER`	`NLS_SORT`
`NLS_LOWER`	`NLS_SORT`
`NLS_INITCAP`	`NLS_SORT`
`NLSSORT`	`NLS_SORT`

Example 7–19 shows how to use NLS parameters in SQL functions.

Example 7–19 Using NLS Parameters in SQL Functions

```
SELECT TO_DATE('1-JAN-99', 'DD-MON-YY',
   'NLS_DATE_LANGUAGE = American') "01/01/99" FROM DUAL;

SELECT TO_CHAR(hire_date, 'DD/MON/YYYY',
   'NLS_DATE_LANGUAGE = French') "Hire Date" FROM employees;

SELECT TO_CHAR(SYSDATE, 'DD/MON/YYYY',
   'NLS_DATE_LANGUAGE = ''Traditional Chinese'' ') "System Date" FROM DUAL;

SELECT TO_CHAR(13000, '99G999D99',
  'NLS_NUMERIC_CHARACTERS = '',.''') "13K" FROM DUAL;

SELECT TO_CHAR(salary, '99G999D99L', 'NLS_NUMERIC_CHARACTERS = '',.''
   NLS_CURRENCY = ''EUR''') salary FROM employees;

SELECT TO_CHAR(salary, '99G999D99C', 'NLS_NUMERIC_CHARACTERS = ''.,''
   NLS_ISO_CURRENCY = Japan') salary FROM employees;

SELECT NLS_UPPER(last_name, 'NLS_SORT = Swiss') "Last Name" FROM employees;

SELECT last_name FROM employees
   ORDER BY NLSSORT(last_name, 'NLS_SORT = German');
```

Note: In some languages, some lowercase characters correspond to more than one uppercase character or some uppercase characters correspond to more than one lowercase characters. As a result, the length of the output from the `NLS_UPPER`, `NLS_LOWER`, and `NLS_INITCAP` functions can differ from the length of the input.

Unacceptable NLS Parameters in SQL Functions

The following NLS parameters are not accepted in SQL functions except for `NLSSORT`:

- `NLS_LANGUAGE`
- `NLS_TERRITORY`
- `NLS_DATE_FORMAT`

The `NLS_DATE_FORMAT` and `NLS_TERRITORY_FORMAT` parameters are not accepted as parameters because they can interfere with required format models. A date format must be specified if an NLS parameter is in a `TO_CHAR` or `TO_DATE` function. As a result, `NLS_DATE_FORMAT` and `NLS_TERRITORY_FORMAT` are not valid NLS parameters for the `TO_CHAR` or `TO_DATE` functions. If you specify `NLS_DATE_FORMAT` or `NLS_TERRITORY_FORMAT` in the `TO_CHAR` or `TO_DATE` function, then an error is returned.

`NLS_LANGUAGE` can interfere with the session value of `NLS_DATE_LANGUAGE`. If you specify `NLS_LANGUAGE` in the `TO_CHAR` function, for example, then its value is ignored if it differs from the session value of `NLS_DATE_LANGUAGE`.

A

Using SQL Command Line

This section provides an introduction to SQL Command Line (SQL*Plus), an interactive and batch command-line query tool that is installed with Oracle Database Express Edition.

This section contains the following topics:

- Overview of SQL Command Line on page A-1
- Using SQL Command Line on page A-1

For information about running SQL language statements, see Chapter 3, "Using SQL".

See Also:

- *SQL*Plus User's Guide and Reference* for complete information about SQL*Plus
- *Oracle Database SQL Reference* for information about using SQL statements
- *Oracle Database Express Edition 2 Day DBA* for information about connecting to Oracle Database XE with SQL Command Line

Overview of SQL Command Line

SQL Command Line (SQL*Plus) is a command-line tool for accessing Oracle Database XE. It enables you to enter and run SQL, PL/SQL, and SQL*Plus commands and statements to:

- Query, insert, and update data
- Execute PL/SQL procedures
- Examine table and object definitions
- Develop and run batch scripts
- Perform database administration

You can use SQL Command Line to generate reports interactively, to generate reports as batch processes, and to write the results to a text file, to a screen, or to an HTML file for browsing on the Internet.

Using SQL Command Line

This section describes SQL Command Line (SQL*Plus), a command-line utility to run SQL and PL/SQL.

This contains the following topics:

Note: Before starting SQL Command Line, make sure that the necessary environmental variables have been set up properly. See *Oracle Database Express Edition 2 Day DBA* for information about setting environmental variables for SQL Command Line.

Starting and Exiting SQL Command Line

To start SQL Command Line from the operating-system command prompt, enter the following:

```
sqlplus
```

When prompted, enter the username and password of the user account (schema) that you want to access in the local database. For example, enter `HR` for the username and *`my_hr_password`* for the password when prompted.

You can also include the username and password when you start SQL Command Line. For example:

```
sqlplus hr/my_hr_password
```

If you want to connect to a database running on a remote system, you need to include a connect string when starting SQL Command Line. For example:

```
sqlplus hr/my_hr_password@host_computer_name
```

After you have started SQL Command Line, the `SQL>` prompt displays as follows:

```
SQL>
```

At the `SQL>` prompt, you can enter SQL statements.

When you want to exit SQL Command Line, enter `EXIT` at the SQL prompt, as follows:

```
SQL> EXIT
```

Displaying Help With SQL Command Line

To display a list of Help topics for SQL Command Line, enter `HELP INDEX` at the SQL prompt as follows:

```
SQL> HELP INDEX
```

From the list of SQL Command Line Help topics, you can display Help about an individual topic by entering `HELP` with a topic name. For example, the following displays Help about the SQL Command Line `COLUMN` command, which enables you to format column output:

```
SQL> HELP COLUMN
```

Entering and Executing SQL Statements and Commands

To enter and execute SQL statements or commands, enter the statement or command at the SQL prompt. At the end of a SQL statement, put a semi-colon (;) and then press the Enter key to execute the statement. For example:

```
SQL> SELECT * FROM employees;
```

If the statement does not fit on one line, enter the first line and press the Enter key. Continue entering lines, and terminate the last line with a semi-colon (;). For example:

```
SQL> SELECT employee_id, first_name, last_name
  2  FROM employees
  3  WHERE employee_id >= 105 AND employee_id <= 110;
```

The output from the previous `SELECT` statement is similar to:

```
EMPLOYEE_ID FIRST_NAME           LAST_NAME
----------- -------------------- -----------------------
        105 David                Austin
        106 Valli                Pataballa
        107 Diana                Lorentz
        108 Nancy                Greenberg
        109 Daniel               Faviet
        110 John                 Chen
6 rows selected.
```

Note that a terminating semi-colon (;) is optional with SQL Command Line commands, such as `DESCRIBE` o r `SET`, but required with SQL statements.

SQL Command Line DESCRIBE Command

SQL Command Line provides the `DESCRIBE` command to display a description of a database object. For example, the following displays the structure of the `employees` table. This description is useful when constructing SQL statements that manipulate the `employees` table.

```
SQL> DESCRIBE employees
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)
```

SQL Command Line SET Commands

The SQL Command Line `SET` commands can be used to specify various SQL Command Line settings, such as the format of the output from SQL `SELECT` statements. For example, the following `SET` commands specify the number of lines for each page and the number of characters for each line in the output:

```
SQL> SET PAGESIZE 200
SQL> SET LINESIZE 140
```

To enable output from PL/SQL blocks with `DBMS_OUTPUT.PUT_LINE`, use the following:

```
SQL> SET SERVEROUTPUT ON
```

To view all the settings, enter the following at the SQL prompt:

```
SQL> SHOW ALL
```

For information about the SQL Command Line `SERVEROUTPUT` setting to display output from a PL/SQL program, see "Inputting and Outputting Data with PL/SQL" on page 4-5.

See Also: *SQL*Plus User's Guide and Reference* for information about setting up the SQL Command Line environment with a login file

Running Scripts From SQL Command Line

You can use a text editor to create SQL Command Line script files that contain SQL*Plus, SQL, and PL/SQL statements. For consistency, use the `.sql` extension for the script file name.

A SQL script file is executed with a `START` or `@` command. For example, in a Windows environment, you can execute a SQL script as follows:

```
SQL> @c:\my_scripts\my_sql_script.sql
```

A SQL script file can be executed in a Linux environment as follows:

```
SQL> START /home/cjones/my_scripts/my_sql_script.sql
```

You can use `SET ECHO ON` to cause a script to echo each statement that is executed. You can use `SET TERMOUT OFF` to prevent the script output from displaying on the screen.

When running a script, you need to include the full path name unless the script is located in the directory from which SQL Command Line was started, or the script is located in the default script location specified by the `SQLPATH` environment variable.

See Also:

- *Oracle Database Express Edition 2 Day DBA* for information about setting environment variables for Oracle Database Express Edition
- *SQL*Plus User's Guide and Reference* for information about setting the SQL Command Line `SQLPATH` environment variable to specify the default location of SQL scripts

Spooling From SQL Command Line

The `SPOOL` command can be used to direct the output from SQL Command Line to a disk file, which enables you to save the output for future review.

To start spooling the output to an operating system file, you enter the `SPOOL` command followed by a file name. For example:

```
SQL> SPOOL my_log_file.log
```

If you want to append the output to an existing file:

```
SQL> SPOOL my_log_file.log APPEND
```

To stop spooling and close a file, enter the following:

```
SQL> SPOOL OFF
```

Using Variables With SQL Command Line

You can create queries that use variables to make `SELECT` statements more flexible. You can define the variable before running a SQL statement, or you specify that the statement prompts for a variable value at the time that the SQL statement is run.

When using a variable in a SQL statement, the variable name must be begin with an ampersand (&).

This section contains the following topics:

- Prompting for a Variable Value in a Query on page A-5
- Reusing a Variable Value in a Query on page A-5
- Defining a Variable Value for a Query on page A-6

For information about using bind variables in PL/SQL code, see "Using Bind Variables With PL/SQL" on page 4-27.

Prompting for a Variable Value in a Query

You can use `&` to identify a variable that you want to define dynamically. In Example A–1, including the `&employee_id` variable causes the SQL statement to prompt for a value when the statement is executed. You can then enter a value for the `employee_id` that corresponds to the employee information that you want to display, such as employee ID 125. Note that you can use any name for the variable, such as `&my_variable`.

Example A–1 Prompting for a Variable Value in SQL Command Line

```
-- prompt for employee_id in a query, you need to enter a valid ID such as 125
SELECT employee_id, last_name, job_id FROM employees
  WHERE employee_id = &employee_id;
```

When you run the previous `SELECT` statement, the output is similar to:

```
Enter value for employee_id: 125
...
EMPLOYEE_ID LAST_NAME                 JOB_ID
----------- ------------------------- ----------
        125 Nayer                     ST_CLERK
```

Reusing a Variable Value in a Query

You can use `&&` to identify a variable that you want to define dynamically multiple times, but only want to prompt the user once. In Example A–2, including the `&&column_name` variable causes the SQL statement to prompt for a value when the statement is executed. The value that is entered is substituted for all remaining occurrences of `&&column_name` in the SQL statement.

Example A–2 Reusing a Variable Value in SQL Command Line

```
-- prompt for a column name, such as job_id, which is then substituted in the
-- remaining identical substitution variables prefixed with &&
SELECT employee_id, last_name, &&column_name FROM employees
  ORDER BY &&column_name;
```

Defining a Variable Value for a Query

In Example A–3, the `&job_id` variable is defined before running the SQL statement with the `DEFINE` command, and the defined value is substituted for the variable when the statement is executed. Because the variable has already been defined, you are not prompted to enter a value.

Example A–3 Defining a Variable for a Query in SQL Command Line

```
-- define a variable value for a query as follows
DEFINE job_id = "ST_CLERK"
-- run a query using the defined value for job_id (ST_CLERK)
SELECT employee_id, last_name FROM employees WHERE job_id = '&job_id';
```

B Reserved Words

This section lists the Oracle Database Express Edition (Oracle Database XE) SQL and PL/SQL reserved words and keywords. You should not use these words to name program or schema objects, such as constants, variables, cursors, columns, tables, or indexes.

The `V$RESERVED_WORDS` data dictionary view provides additional information about all keywords, including whether the keyword is always reserved or is reserved only for particular uses. For more information, refer to *Oracle Database Reference*.

This section contains the following topics:

- SQL Reserved Words on page B-1
- PL/SQL Reserved Words on page B-2

SQL Reserved Words

This section lists Oracle Database XE SQL Command Line reserved words. Words followed by an asterisk (*) are also ANSI reserved words.

Table B–1 lists the SQL reserved words.

Table B–1 SQL Reserved Words

Begins with:	Reserved Words
A	ACCESS, ADD*, ALL*, ALTER*, AND*, ANY*, AS*, ASC*, AUDIT
B	BETWEEN*, BY*
C	CHAR*, CHECK*, CLUSTER, COLUMN, COMMENT, COMPRESS, CONNECT*, CREATE*, CURRENT*
D	DATE*, DECIMAL*, DEFAULT*, DELETE*, DESC*, DISTINCT*, DROP*
E	ELSE*, EXCLUSIVE, EXISTS
F	FILE, FLOAT*, FOR*, FROM*
G	GRANT*, GROUP*
H	HAVING*
I	IDENTIFIED, IMMEDIATE*, IN*, INCREMENT, INDEX, INITIAL, INSERT*, INTERSECT*, INTO*, IS*
L	LEVEL*, LIKE*, LOCK, LONG
M	MAXEXTENTS, MINUS, MLSLABEL, MODE, MODIFY
N	NOAUDIT, NOCOMPRESS, NOT*, NOWAIT, NULL*, NUMBER
O	OF*, OFFLINE, ON*, ONLINE, OPTION*, OR*, ORDER*
P	PCTREE, PRIOR*, PRIVLEGES*, PUBLIC*
R	RAW, RENAME, RESOURCE, REVOKE*, ROW, ROWID, ROWNUM, ROWS*

Table B–1 (Cont.) SQL Reserved Words

Begins with:	Reserved Words
S	SELECT*, SESSION*, SET*, SHARE, SIZE*, SMALLINT*, START, SUCCESSFUL, SYNONYM, SYSDATE
T	TABLE*, THEN*, TO*, TRIGGER
U	UID, UNION*, UNIQUE*, UPDATE*, USER*
V	VALIDATE, VALUES*, VARCHAR*, VARCHAR2, VIEW*
W	WHENEVER*, WHERE, WITH*

PL/SQL Reserved Words

The words listed in this section are reserved by PL/SQL. Some of these words are also reserved by SQL.

These words reserved by PL/SQL are classified as keywords or reserved words. See Table B–2 and Table B–3. Reserved words can never be used as identifiers. Keywords can be used as identifiers, but this is not recommended.

Table B–2 lists the PL/SQL reserved words.

Table B–2 PL/SQL Reserved Words

Begins with:	Reserved Words
A	ALL, ALTER, AND, ANY, ARRAY, ARROW, AS, ASC, AT
B	BEGIN, BETWEEN, BY
C	CASE, CHECK, CLUSTERS, CLUSTER, COLAUTH, COLUMNS, COMPRESS, CONNECT, CRASH, CREATE, CURRENT
D	DECIMAL, DECLARE, DEFAULT, DELETE, DESC, DISTINCT, DROP
E	ELSE, END, EXCEPTION, EXCLUSIVE, EXISTS
F	FETCH, FORM, FOR, FROM
G	GOTO, GRANT, GROUP
H	HAVING
I	IDENTIFIED, IF, IN, INDEXES, INDEX, INSERT, INTERSECT, INTO, IS
L	LIKE, LOCK
M	MINUS, MODE
N	NOCOMPRESS, NOT, NOWAIT, NULL
O	OF, ON, OPTION, OR, ORDER,OVERLAPS
P	PRIOR, PROCEDURE, PUBLIC
R	RANGE, RECORD, RESOURCE, REVOKE
S	SELECT, SHARE, SIZE, SQL, START, SUBTYPE
T	TABAUTH, TABLE, THEN, TO, TYPE
U	UNION, UNIQUE, UPDATE, USE
V	VALUES, VIEW, VIEWS
W	WHEN, WHERE, WITH

Table B–3 lists the PL/SQL keywords.

Table B–3 PL/SQL Keywords

Begins with:	Keywords
A	A, ADD, AGENT, AGGREGATE, ARRAY, ATTRIBUTE, AUTHID, AVG
B	BFILE_BASE, BINARY, BLOB_BASE, BLOCK, BODY, BOTH, BOUND, BULK, BYTE
C	C, CALL, CALLING, CASCADE, CHAR, CHAR_BASE, CHARACTER, CHARSETFORM, CHARSETID, CHARSET, CLOB_BASE, CLOSE, COLLECT, COMMENT, COMMIT, COMMITTED, COMPILED, CONSTANT, CONSTRUCTOR, CONTEXT, CONVERT, COUNT, CURSOR, CUSTOMDATUM
D	DANGLING, DATA, DATE, DATE_BASE, DAY, DEFINE, DETERMINISTIC, DOUBLE, DURATION
E	ELEMENT, ELSIF, EMPTY, ESCAPE, EXCEPT, EXCEPTIONS, EXECUTE, EXIT, EXTERNAL
F	FINAL, FIXED, FLOAT, FORALL, FORCE, FUNCTION
G	GENERAL
H	HASH, HEAP, HIDDEN, HOUR
I	IMMEDIATE, INCLUDING, INDICATOR, INDICES, INFINITE, INSTANTIABLE, INT, INTERFACE, INTERVAL, INVALIDATE, ISOLATION
J	JAVA
L	LANGUAGE, LARGE, LEADING, LENGTH, LEVEL, LIBRARY, LIKE2, LIKE4, LIKEC, LIMIT, LIMITED, LOCAL, LONG, LOOP
M	MAP, MAX, MAXLEN, MEMBER, MERGE, MIN, MINUTE, MOD, MODIFY, MONTH, MULTISET
N	NAME, NAN, NATIONAL, NATIVE, NCHAR, NEW, NOCOPY, NUMBER_BASE
O	OBJECT, OCICOLL, OCIDATETIME, OCIDATE, OCIDURATION, OCIINTERVAL, OCILOBLOCATOR, OCINUMBER, OCIRAW, OCIREFCURSOR, OCIREF, OCIROWID, OCISTRING, OCITYPE, ONLY, OPAQUE, OPEN, OPERATOR, ORACLE, ORADATA, ORGANIZATION, ORLANY, ORLVARY, OTHERS, OUT, OVERRIDING
P	PACKAGE, PARALLEL_ENABLE, PARAMETER, PARAMETERS, PARTITION, PASCAL, PIPE, PIPELINED, PRAGMA, PRECISION, PRIVATE
R	RAISE, RANGE, RAW, READ, RECORD, REF, REFERENCE, REM, REMAINDER, RENAME, RESULT, RETURN, RETURNING, REVERSE, ROLLBACK, ROW
S	SAMPLE, SAVE, SAVEPOINT, SB1, SB2, SB4, SECOND, SEGMENT, SELF, SEPARATE, SEQUENCE, SERIALIZABLE, SET, SHORT, SIZE_T, SOME, SPARSE, SQLCODE, SQLDATA, SQLNAME, SQLSTATE, STANDARD, STATIC, STDDEV, STORED, STRING, STRUCT, STYLE, SUBMULTISET, SUBPARTITION, SUBSTITUTABLE, SUBTYPE, SUM, SYNONYM
T	TDO, THE, TIME, TIMESTAMP, TIMEZONE_ABBR, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TRAILING, TRANSAC, TRANSACTIONAL, TRUSTED, TYPE
U	UB1, UB2, UB4, UNDER, UNSIGNED, UNTRUSTED, USE, USING
V	VALIST, VALUE, VARIABLE, VARIANCE, VARRAY, VARYING, VOID
W	WHILE, WORK, WRAPPED, WRITE
Y	YEAR
Z	ZONE

C

Using a PL/SQL Procedure With PHP

This section provides an example of the use of a PL/SQL stored procedure with PHP.

This section does not provide detailed information about PHP or its use with Oracle Database Express Edition. For a brief summary of PHP and links to resources for PHP, see "PHP" on page 1-9.

This section contains the following topics:

- PHP and Oracle Database XE on page C-1
- Creating a PHP Program That Calls a PL/SQL Stored Procedure on page C-1

PHP and Oracle Database XE

PHP is a widely-used, open-source, interpretive, HTML-centric, server-side scripting language. PHP is especially suited for Web development and can be embedded into HTML pages. Zend Core for Oracle, developed in partnership with Zend Technologies, enables application development using PHP with Oracle Database XE.

To run the PHP program in Example C–1 on page C-2, you need to have Oracle Database XE, Apache 1.3.x or later, and Zend Core for Oracle installed on your computer.

See Also:

- *Oracle Database Express Edition 2 Day Plus PHP Developer Guide* for information about application development using Zend Core for Oracle and Oracle Database XE
- PHP Development Center at

  `http://www.oracle.com/technology/tech/php/index.html`
- Zend Core for Oracle at

  `http://www.oracle.com/technology/tech/php/zendcore/ind`
  `ex.html`

Creating a PHP Program That Calls a PL/SQL Stored Procedure

Example C–1 shows the PHP code that searches for and displays employee information based on the first and last name of an employee. The PHP program first gathers user input for the first and last name of an employee. The text input can be the full names or substrings of the first and last names of the employee. With valid input, a connection is made to Oracle Database XE, the `get_emp_info` procedure is called to search for the employee records that match the input strings, and then the results are displayed.

The PL/SQL `get_emp_info` procedure is created in Example 5–15 on page 5-22 and it determines the result set for the cursor variable (`REF CURSOR`) that is passed to the PHP program. The package specification in Example 5–13 on page 5-21 defines the cursor variable (`my_refcur_typ`) that is declared in the `get_emp_info` procedure. A cursor variable can be passed as a parameter to other packages, procedures, and functions. For information about cursor variables (`REF CURSOR`s) see "Cursor Variables (REF CURSORs)" on page 4-22. For information about using types in package specifications, see "Accessing Types in Packages" on page 5-21.

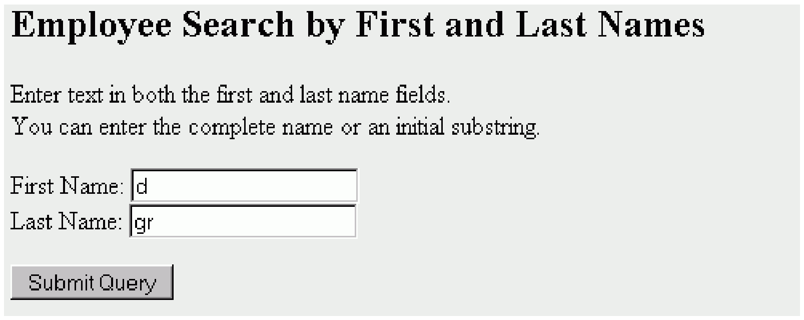
The PHP program in Example C–1 is intended only to be an illustration of the use of a PL/SQL stored procedure with PHP. It does not include error checking or many other PHP features.

Save the PHP program in Example C–1 as `emp_search.php`. Before running the PHP program in Example C–1, the PL/SQL `get_emp_info` procedure in Example 5–15 on page 5-22 must be created by the `HR` user.

Example C–1 Creating a PHP Program for Use With a PL/SQL Procedure

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
       "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>Search for Employee Information</title>
</head>
<body bgcolor="#EEEEEE">
<h2>Employee Search by First and Last Names</h2>

<?php

function get_input($first = "", $last = "")
{
  echo <<<END
  <form action="emp_search.php" method="post">
  First Name:
  <input type="text" name="first" value="$first">
  <br>
  Last Name:
  <input type="text" name="last" value="$last">
  <p>
  <input type="submit">
  </form>
END;
}

if(!isset($_REQUEST['first'])) {
   echo "Enter text in both the first and last name fields.<br>
         You can enter the complete name or an initial substring.<p>";
   get_input();
}
else {
  // check whether the input fields are empty before continuing
  if (empty($_REQUEST['first']) or empty($_REQUEST['last'])) {
    echo "You did not enter text in both
          fields, please re-enter the information.<p>";
    get_input($_REQUEST['first'], $_REQUEST['last']);
  }
  else {
    // if text has been entered in both input fields, then
    // create a database connection to Oracle XE using
    // password hr for user HR with a local connection to the XE database
    $conn = oci_connect('hr', 'hr', '//localhost/XE');

    // execute the function that calls the PL/SQL stored procedure
    $emp = get_employees($conn, $_REQUEST['first'], $_REQUEST['last']);

    // display results
    print_results($emp, 'Employee Information');

    // close the database connection
    oci_close($conn);
  }
}

// this functions calls a PL/SQL procedure that uses a ref cursor to fetch records
function get_employees($conn, $firstname, $lastname)
{
  // execute the call to the stored PL/SQL procedure
  $sql = "BEGIN get_emp_info(:firstname, :lastname, :refcur); END;";
  $stmt = oci_parse($conn, $sql);

  // bind the first and last name variables
  oci_bind_by_name($stmt, ':firstname', $firstname, 20);
  oci_bind_by_name($stmt, ':lastname', $lastname, 25);

  // bind the ref cursor
  $refcur = oci_new_cursor($conn);
  oci_bind_by_name($stmt, ':REFCUR', $refcur, -1, OCI_B_CURSOR);

  // execute the statement
  oci_execute($stmt);

  // treat the ref cursor as a statement resource
  oci_execute($refcur, OCI_DEFAULT);
  oci_fetch_all($refcur, $employeerecords, null, null, OCI_FETCHSTATEMENT_BY_ROW);

  // return the results
  return ($employeerecords);
}

// this function prints information in the returned records
function print_results($returned_records, $report_title)
{
  echo '<h3>'.htmlentities($report_title).'</h3>';
  if (!$returned_records) {
    echo '<p>No Records Found</p>';
  }
  else {
    echo '<table border="1">';
    // print one row for each record retrieved
    // put the fields of each record in separate table cells
    foreach ($returned_records as $row) {
      echo '<tr>';
      foreach ($row as $field) {
        print '<td>'.
             ($field ? htmlentities($field) : '&nbsp;').'</td>';
      }
    }
    echo '</table>';
  }
}

?>

</body>
</html>
```

After you save the PHP program (`emp_search.php`) and access the file in a Web browser, you are prompted to enter a first and last name. After you enter text for the first and last name, click the Submit Query button.

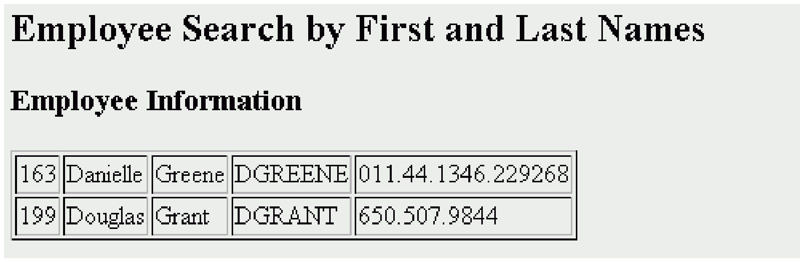


Employee Search by First and Last Names

Enter text in both the first and last name fields.
You can enter the complete name or an initial substring.

First Name: d
Last Name: gr

If you have entered text in both the first and last name fields, the employees table is searched and the results are displayed using the first version of the `get_emp_info` procedure shown in Example 5–15 on page 5-22.

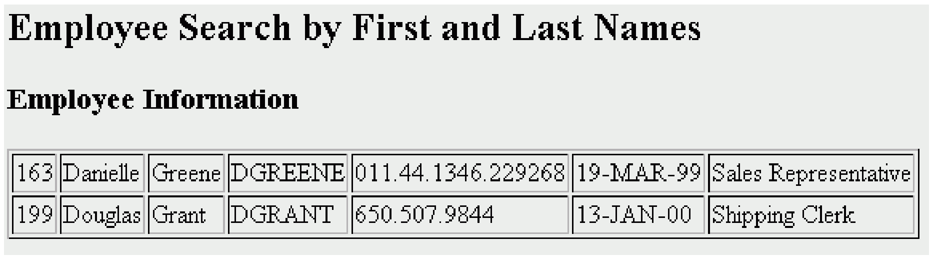


Employee Search by First and Last Names

Employee Information

163	Danielle	Greene	DGREENE	011.44.1346.229268
199	Douglas	Grant	DGRANT	650.507.9844

After you have updated the `get_emp_info` procedure shown in Example 5–15 on page 5-22, you can run the PHP `emp_search.php` program again and check the results. If you have entered text in both the first and last name fields, the employees table is searched and the results are displayed using the updated procedure.



Employee Search by First and Last Names

Employee Information

163	Danielle	Greene	DGREENE	011.44.1346.229268	19-MAR-99	Sales Representative
199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-00	Shipping Clerk

D

Using a PL/SQL Procedure With JDBC

This section provides an example of the use of a PL/SQL stored procedure with JDBC.

This section does not provide detailed information about Java and JDBC, or their use with Oracle Database Express Edition. For a brief summary of JDBC and links to resources for JDBC, see "Oracle Java Database Connectivity (JDBC)" on page 1-8.

This section contains the following topics:

- JDBC and Oracle Database XE on page D-1
- Creating a Java Program That Calls a PL/SQL Procedure on page D-1

JDBC and Oracle Database XE

Oracle Java Database Connectivity (JDBC) is an API that enables Java to send SQL statements to an object-relational database such as Oracle Database XE.

To run the Java program in Example D–1 on page D-2, you need to have Oracle Database XE and the full Java 2 Software Development Kit, Standard Edition (J2SE SDK), installed on your computer. Note that the Oracle Database XE Client is included in Oracle Database Express Edition.

See Also:

- *Oracle Database Express Edition 2 Day Plus Java Developer Guide* for information about using Java to access and modify data in Oracle Database XE
- Information about the JDBC API at

  `http://java.sun.com/products/jdbc/index.jsp`
- Information about the Oracle JDBC Drivers at

  `http://www.oracle.com/technology/tech/java/sqlj_jdbc/index.html`
- Information about installing Java at

  `http://java.sun.com/j2se/index.jsp`

Creating a Java Program That Calls a PL/SQL Procedure

Example D–1 shows the Java code that searches for and displays employee information based on the first and last names of an employee. The Java program accepts user input for the first and last names from command line arguments. The arguments can be the full names or substrings of the first and last names of the employee. When the program is executed, a connection is made to Oracle Database

XE, the `get_emp_info` procedure is called to search for the employee records that match the input strings, and then the results are displayed.

The PL/SQL `get_emp_info` procedure is created in Example 5–15 on page 5-22. The procedure determines the result set for the cursor variable (`REF CURSOR`) that is passed to the Java program. The package specification in Example 5–13 on page 5-21 defines the cursor variable (`my_refcur_typ`) that is declared in the `get_emp_info` procedure. A cursor variable can be passed as a parameter to other packages, procedures, and functions. For information about cursor variables (`REF CURSOR`s) see "Cursor Variables (REF CURSORs)" on page 4-22. For information about using types in package specifications, see "Accessing Types in Packages" on page 5-21.

The Java program in Example D–1 is intended only to be an illustration of the use of a PL/SQL stored procedure with JDBC. It does not include error checking or many other Java features.

Save the Java program in Example D–1 as `EmpSearch.java`. Before running the Java program in Example D–1, the PL/SQL `get_emp_info` procedure in Example 5–15 on page 5-22 must be created by the `HR` user.

Example D–1 Creating a Java Program for Use With a PL/SQL Procedure

```
import java.sql.*;
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.*;

public class EmpSearch
{

  public static void main (String args[]) throws SQLException
  {
   // check whether there are two command-line arguments before proceeding
   if ( args.length < 2)
    {
     System.out.println("Enter both a first and last name as command-line arguments.");
     System.out.println("You can enter a complete name or an initial substring.");
     System.out.println("For example: java EmpSearch j doe");
     }
   else
     {
      // connect to a local XE database as user HR
      OracleDataSource ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:hr/hr@localhost:1521/XE");
      Connection conn = ods.getConnection();

      // call the PL/SQL procedures with the three parameters
      // the first two string parameters (1 and 2) are passed to the procedure
      // as command-line arguments
      // the REF CURSOR parameter (3) is returned from the procedure
      String jobquery = "begin get_emp_info(?, ?, ?); end;";
      CallableStatement callStmt = conn.prepareCall(jobquery);
      callStmt.registerOutParameter(3, OracleTypes.CURSOR);
      callStmt.setString(1, args[0]);
      callStmt.setString(2, args[1]);
      callStmt.execute();

      // return the result set
      ResultSet rset = (ResultSet)callStmt.getObject(3);

      // determine the number of columns in each row of the result set
      ResultSetMetaData rsetMeta = rset.getMetaData();
      int count = rsetMeta.getColumnCount();

      // print the results, all the columns in each row
      while (rset.next()) {
          String rsetRow = "";
          for (int i=1; i<=count; i++){
                 rsetRow = rsetRow + " " + rset.getString(i);
          }
          System.out.println(rsetRow);
       }

    }
  }
}
```

Before you attempt to compile the Java program, make sure the `CLASSPATH` has been set. For example, on a Windows computer you could use the following value for `CLASSPATH`, if Oracle Database XE is installed in the default location.

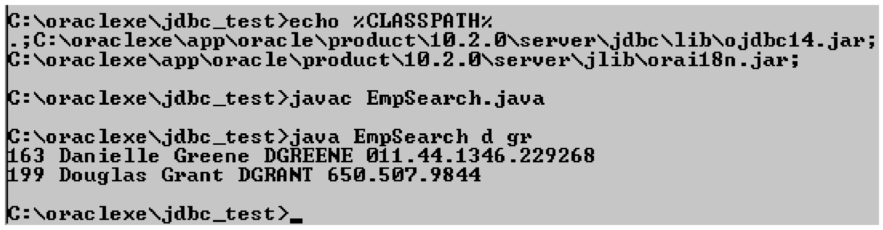
```
.;C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;C:\oraclexe\app\oracle\
product\10.2.0\server\jlib\orai18n.jar;
```

Compile the `EmpSearch.java` Java program. For example, on a Windows computer you could use one of the following examples, depending on whether the path to `javac` has been set.

```
javac EmpSearch.java
c:\j2sdk1.4.2_04\bin\javac EmpSearch.java
```

Run the program with valid command-line arguments. For example, on a Windows computer you could use one of the following examples, depending on whether the path to `java` has been set. With the `d` and `gr` command-line arguments, the PL/SQL procedure searches for employees whose first name starts with `d` and last name starts with `gr`.

```
java EmpSearch d gr
c:\j2sdk1.4.2_04\bin\java EmpSearch d gr
```



```
C:\oraclexe\jdbc_test>echo %CLASSPATH%
.;C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;
C:\oraclexe\app\oracle\product\10.2.0\server\jlib\orai18n.jar;

C:\oraclexe\jdbc_test>javac EmpSearch.java

C:\oraclexe\jdbc_test>java EmpSearch d gr
163 Danielle Greene DGREENE 011.44.1346.229268
199 Douglas Grant DGRANT 650.507.9844

C:\oraclexe\jdbc_test>_
```

Index