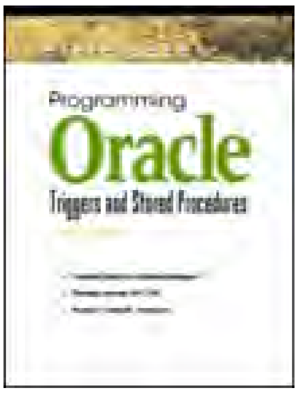


Programming Oracle® Triggers and Stored Procedures, Third Edition

By Kevin Owens

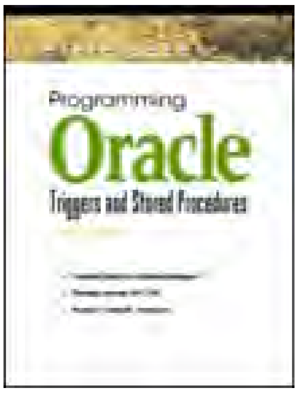
Publisher: Prentice Hall PTR
Pub Date: December 05, 2003
ISBN: 0-13-085033-0
Pages: 448

Effectively create and manage complex databases with Oracle! Systems and database expert Kevin Owens explores PL/SQL, Oracle's answer to the Structured Query Language (SQL), and teaches you what you need to know to build robust and complex databases for your business.

Using easy-to-follow instructions and examples, this book presents techniques to take advantage of Oracle features such as triggers and stored procedures-features that allow your databases to incorporate business rules which are easy to manage and modify as the business evolves. Topics covered include:

- Viewing constraints in the data dictionary
- Complex rule enforcement
- PL/SQL program units and language features
- Data types and composite structure
- Error handling and exceptions
- Inter-process communications
- Declarative constraints, including primary key, unique, foreign key, check, and much more

Programming Oracle Triggers and Stored Procedures, Third Edition, is an invaluable resource for database developers, designers, and project leaders looking to build and maintain truly intelligent, complex databases.



• Table of Contents

Programming Oracle® Triggers and Stored Procedures, Third Edition

By Kevin Owens

Start Reading

Publisher: Prentice Hall PTR
Pub Date: December 05, 2003
ISBN: 0-13-085033-0
Pages: 448

Copyright

Library of Congress Cataloging-in-Publication Data

Owens, Kevin.
  Programming Oracle triggers and stored procedures / Kevin Owens.
    p. cm.
  ISBN 0-13-085033-0
  1. Oracle (Computer file) 2. Relational databases. 3. Computer
software--Development. I. Title.

  QA76.9.D3O97 2004
  005.75'85--dc22
                                        2003023685

Editorial/production supervision: Jessica Balch (Pine Tree Composition, Inc.)

Cover design director: Jerry Votta

Cover design: Anthony Gemmellaro

Art director: Gail Cocker-Bogusz

Manufacturing manager: Alexis Heydt-Long

Publisher: Jeff Pepper

Editorial assistant: Linda Ramagnano

Marketing manager: Robin O'Brien

Full-service production manager: Anne R. Garcia

About Prentice Hall Professional Technical Reference

With origins reaching back to the industry's first computer science publishing program in the 19605, and formally launched as its own imprint in 1986, Prentice Hall Professional Technical Reference (PH PTR) has developed into the leading provider of technical books in the world today. Our editors now publish over 200 books annually, authored by leaders in the fields of computing, engineering, and business.

Our roots are firmly planted in the soil that gave rise to the technical revolution. Our bookshelf contains many of the industry's computing and engineering classics: Kernighan and Ritchie's *C Programming Language*, Nemeth's *UNIX System Administration Handbook*, Horstmann's *Core Java*, and Johnson's *High-Speed Digital Design*.

PH PTR acknowledges its auspicious beginnings while it looks to the future for inspiration. We continue to evolve and break new ground in publishing by providing today's professionals with tomorrow's solutions.

Preface

This book emphasizes core concepts of the Oracle Database Server: database tables, indexes, tablespaces, constraints, triggers, and PL/SQL. Mastering these core components is essential to applications development. Whether you are a web developer or a client/server or backend programmer, the content in this book will help you realize the potential of implementing logic in the database server with declarative and procedural constraints, using PL/SQL triggers and stored procedures.

Who Will Benefit from This Book?

This is a programmer's book. Some programming knowledge is helpful but not essential. No prior knowledge of Oracle is required. Developers transitioning from other databases to an Oracle environment can benefit from the detailed explanations, numerous figures, and many code examples that illustrate Oracle structures. The discussion on triggers and PL/SQL is geared toward an individual who wants a quick transition from other programming paradigms to the PL/SQL model. I have worked on Oracle projects staffed with non-Oracle database developers and understand the confusion with terminology. A consistent focus of this text is to explain the basic paradigm, which helps readers who are new to the technology as well as experienced developers who are cross-training.

This text can help developers who are transitioning from non-relational databases to Oracle. Chapter 1 addresses the beginnings of SQL. Developers from hierarchical and network databases will find this chapter a place in which that they can put SQL into perspective. The early part of my career included five years on a hierarchical database and three years on a network database. I understand the transition needed to switch to Oracle, and the explanations in this text should address the needs of non-relational developers who are cross-training.

Developers who naturally "dig" into a subject will benefit from the discussion on the Oracle data dictionary, addressed extensively in Chapter 5. This chapter illustrates extracting constraint definitions from the data dictionary. A curious developer can expand this simple technique to extract unlimited information from the data dictionary. This is highly beneficial to anxious learners. Readers of the previous editions have mentioned this as one of the most helpful parts of the book because it taught them how to explore and learn on their own.

Developers often find themselves performing database administration—usually on a development server. This text is a good starting point for beginning database administration. Developers are naturally in touch with the logical world of tables and tablespaces. The discussion on the physical aspects of the database, tablespace data files, is an excellent starting point for a developer who needs to understand the physical aspects of an Oracle environment.

Foremost is the emphasis on building business logic in an Oracle database. A consistent theme in the text is to make the database work for you. An application team that capitalizes on declarative constraints can produce an "intelligent" database in which all data conforms to strict business-rule logic. Furthermore, when the team incorporates triggers with PL/SQL, the database integrity extends to a point where it can perform logical operations on the end-user's behalf. This can include complex business-rule enforcement as well as actions such as automatic email notification, which is illustrated in Chapter 11.

How Is This Book Structured?

Chapters 1 and 2

Chapters 1 and 2 cover Oracle tables, column types, and how to use SQL*Plus to interact with the database. Some highly useful features of SQL*Plus are described including SQL*Plus commands to show a SQL execution plan and its statistics. These chapters concentrate on terminology, explaining the intuitive meaning of the terms SQL, SQL*Plus, and PL/SQL.

Chapters 3, 4, and 5

Chapters 3, 4, and 5 thoroughly cover declarative constraints including extracting constraint definitions from the Oracle data dictionary. These chapters (a) provide you with the knowledge to build a database with high integrity—something every client wants, (b) empower you with the skill to extract a wide variety of information from the data dictionary—a skill everyone needs, and (c) provide insight into declarative constraints and help you understand how they work.

Chapters 6, 7, and 8

Chapters 6, 7, and 8 cover database triggers, also called procedural constraints. Chapters 6 and 7 provide an intuitive understanding with numerous figures to illustrate table row and statement level triggers. This intuitive understanding is critical in helping you choose what type of trigger you should write. Chapter 8 illustrates the use of database triggers and PL/SQL for complex rule enforcement.

Chapter 9

Chapter 9 helps you get started quickly with compiling PL/SQL programs. A rapid learning approach is to code while you learn. This chapter helps you understand the PL/SQL coding environment. After completing this chapter, you will have a basic understanding of how to code, compile, and execute PL/SQL procedures. The following two chapters provide the technical content of PL/SQL while this chapter introduces the means to code and test as you learn.

Chapter 10

Chapter 10 looks at PL/SQL from a software engineering perspective. It is not enough just to know the PL/SQL syntax. Before a language bears meaning, one needs to understand the PL/SQL paradigm and the model for PL/SQL program units: the procedure, function, and package. This chapter teaches you PL/SQL program units conceptually and encourages you to think about when and how to use these program units during the design phase.

Chapter 11

Chapter 11 covers PL/SQL language features. Particular attention is given to PL/SQL constructs and built-in SQL functions, emphasizing writing simple and easy-to-understand algorithms. The chapter concludes with a section that illustrates an application using a database trigger to send email notification using the Oracle built-in Alerts package.

All scripts in the text were run against Oracle 9*i* Release 2 and 10*g* Version 10.1.0.

Conventions

The use of upper and lower case is intended to make clear the identity of SQL and PL/SQL keywords. All SQL and PL/SQL code is written in lower case except for language keywords, which are upper case. The following, taken from Chapter 3, illustrates this writing style.

```
CREATE SEQUENCE student_vehicles_pk_seq;
```

Quoted expressions are often in upper case as well as all references to Oracle built-in functions. The following, taken from Chapter 11, includes references to two Oracle built-in functions (NEWTIME and SYSDATE) written in upper case.

```
pst_date := NEW_TIME(SYSDATE, 'EST','PST');
```

Oracle provides may built-in packages. The sample code in this text refers to all built-in packages in lower case. This decision was made to make the code more readable and easier on the eyes. Excessive upper case text can be rough on the corneas. The following, from Chapter 11, illustrates the choice of lower case when writing references to the Oracle built-in package DBMS_ALERT.

```
dbms_alert.register('OUT_OF_STOCK_ITEM');
```

The PL/SQL code examples often include counter variable declarations; these are often in upper case. The following, from Chapter 11, declares a variable, ID. This is not a keyword but is easier to read in upper case.

```
ID   VARCHAR2(10),
```

Finally non-code text, that is, the paragraphs of the text, always refers to code items in upper case. The text of a paragraph may refer to a constraint name or PL/SQL variable name, and always, the code identifier is written in upper case within the body of a paragraph.

Acknowledgments

I would like to thank the technical review and editing team: Pat Starr, Kathryn Tyser, and Linda Owens. Thanks also to Jeffrey Pepper at Prentice Hall for diligence, and the thorough editing from copyeditor Laura Burgess, and Jessica Balch at Pine Tree Composition.

Kevin Owens

Chapter One. An Introduction to Relational Database Tables

1.1 Before Relational Tables

In 1969, Apollo astronauts Neil Armstrong and Buzz Aldrin walked on the moon. That same year Dr. Edgar Frank Codd, Ph.D., developed the theory for relational database systems. Shortly after this development, Codd published "A Relational Model of Data for Large Shared Data Banks" in the Association of Computing Machinery (ACM) journal, *Communications of the ACM*. Codd's model became the foundation for a prototype; a relational database project within IBM called System/R. Studies were performed at the IBM San Jose Research Center (now the Almaden Research Center) and by 1974 IBM had a running prototype relational database system. This system was based on multitable queries and multiuser access. The method for accessing data in System/R was called Structured English Query Language (SEQUEL).

Following its research and development, IBM released System/R as a prototype. It was used by the MIT Sloan School of Management and commercial organizations within the manufacturing and inventory sectors. The project demonstrated that Codd's theories could be applied and implemented in the real world, but ended in 1979.

The success of System/R did not receive quite the same attention the world gave the Apollo astronauts on July 20, 1969; but, System/R's success created a new paradigm in the world of information technology—a paradigm that dominates the information age to this day.

Codd believed that programmers should be able to control the exact data used to join and construct queries from multiple independent tables. This was not the case with existing hierarchical and network databases. For these other technologies, an application programmer would code an algorithm to navigate through the database by accessing data records based on the predefined navigation paths. As software requirements changed, programmers frequently needed database modifications to these paths and new navigations paths needed to be defined. The software development process was slow and impaired with frequent meetings on data access. Codd's model had a broad impact on the software development process.

System/R was implemented by a group of highly talented and educated software engineers. They demonstrated that large information systems could be built on Codd's theory of multi-independent tables and a structured query language. This was the birth of SEQUEL. Throughout the System/R project, engineers openly published their work in technical journals and at international conferences. The policy of open publication is considered to have been a factor in the success of System/R—leading to the creation of many relational databases such as DB2 and Oracle.

IBM passed control of the System/R structured query language to the American National Standards Institute (ANSI), a private nonprofit organization that administers U.S. voluntary technologies. Because of trademark laws, the language was renamed from SEQUEL to SQL for "Structured Query Language." Today SQL is an ANSI standard and an International Standards Organization (ISO) standard.

In 1977, a group of engineers who had been following the System/R project, primarily through publications, recognized its potential and formed a company. This company, Software Development Laboratories, was later named Relational Software. This new company developed and marketed the first commercial version of a relational database management system, which was based on the ANSI SQL Standard. They named their product Oracle.

When project System/R ended in 1979, many information systems relied on either hierarchical or network database architecture. In the 1980s, many development projects moved toward relational technology. Today, the majority of new information systems are relational; however, hierarchical and network database products have still maintained a market presence.

IMS/DB is the flagship hierarchical database of IBM. Rockwell International and IBM developed IMS in 1969 to manage data for the National Aeronautics and Space Administration's (NASA's) Apollo program. IMS/DB continues today with a solid international presence. It is industry-wide and serves an estimated 200 million end users. The IMS/DB database served the Apollo program including Apollo 11, which sent Armstrong, Aldrin, and pilot Mike Collins to the moon.

Computer Associates' network database product is CA-IDMS, a widely used database serving 2,000 sites around the world. The product has changed in name and ownership but continues to support large corporate data systems that run on IBM OS/390 platforms.

IDMS has an interesting history. John J. Cullinane founded Cullinet Software, Inc. in 1968. He bought the rights to IDMS, a CODASYL database, from B.F. Goodrich. CODASYL, an acronym for Conference on Data System Languages, was established in 1960 by the Department of Defense for the purpose of standardizing languages and software applications. Cullinet was the first software company to be listed on the New York Stock Exchange. Computer Associates purchased Cullinet in September of 1989 and renamed the product CA-IDMS.

Although CA-IDMS is based on database network architecture, the product includes an ANSI-compliant SQL Option. This option allows applications to access IDMS data using SQL. A set of comprehensive tools is available that includes agents for enterprise monitoring, parallel transaction processing using IBM Parallel Sysplex Cluster Technology, JDBC support for Java, and many other eBusiness technologies. The majority of IDMS applications are written in COBOL.

Oracle, predominately known for the Relational Enterprise Server, owns two additional database products: (a) a network database product called DBMS, which is a CODASYL database, and (b) a relational database product called Rdb. Both Rdb and DBMS were purchased from Digital Equipment Corporation (DEC) prior to the acquisition of DEC by Compaq.

Oracle acquired DBMS when it purchased the Rdb product family from DEC. Ken Olsen, an MIT graduate, his brother Stan, and Harlan Anderson founded DEC. In addition to hardware architectures such as the PDP-11, VAX, and Alpha, DEC developed Rdb, a relational database, and also DBMS, a network database. The network database was renamed from VAX/DBMS to DEC DBMS when version V5.0 was released on the Alpha AXP platform.

Oracle's CODASYL DBMS is a multi-user network database that, like Rdb, is optimized for the Compaq OpenVMS operating system on either an Alpha or VAX architecture platform. DBMS is highly suited for manufacturing systems and shop floor systems that require stable environments where database information is fairly static. Consillium Corporation's WorkStream is the most widely installed Manufacturing Execution System (MES) in the semiconductor and electronics industries. WorkStream has a long history of running on VAX/DBMS and today runs primarily on VAX and Alpha servers using Oracle CODASYL DBMS.

Rdb7 was Oracle's first release of Rdb that was conceived and engineered as an Oracle product. Oracle's Rdb7 is an enterprise relational database optimized for digital platforms; that is, the Hewlett-Packard Corporation's OpenVMS operating system and Compaq Digital UNIX. When Compaq acquired DEC it replaced the DEC version of UNIX, called Ultrix, with the product name DIGITAL UNIX—an operating system based on a 64-bit architecture that runs on Alpha AXP platforms.

In summary, the database market is clearly dominated by relational technology; however, hierarchical and network databases continue to meet the needs of enterprise network computing environments. IBM IMS/DB is a dominant hierarchical database that can service high-transaction rates and has a large install base. Two network databases—CA-IDMS, licensed by Computer Associates, and DBMS, licensed by Oracle,—play critical roles in support of high-transaction enterprise-wide systems. In addition to the Oracle Enterprise Server, Oracle Corporation also owns Rdb7, a relational database, and DBMS, a network database—both products are optimized for Compaq platforms, including Alpha AXP. Today the Oracle Database Server addresses the needs of small and large businesses that require information processing for online transaction processing (OLTP) systems, decision support systems (DSS), plus the eBusiness solutions. These types of applications can operate on small, stand-alone servers, or on distributed, high-availability architectures.

[Team LiB] PREVIOUS NEXT

1.2 SQL

SQL means something very specific, whereas the term *database* has broad meaning and interpretation in everyday conversation. This section elaborates on the SQL language as a standard. Topics include SQL implementation, embedded SQL, Direct SQL, and SQL*Plus as an implementation of Direct SQL.

1.2.1 ANSI Standard

SQL is a language currently defined in an ANSI standard (the previous section discussed the history of SQL as it passed from IBM to ANSI). If you are writing an Oracle application that uses SQL, then you are coding SQL statements whose syntax complies with ANSI-X3.135-1999, "Database Language SQL."

This ANSI standard is available to the public through the ANSI Web store. You can visit the site, *http://webstore.ansi.org/ansidocstore/*, which has a search form field. Using the search field, you can find SQL standards with a search on "ANSI X3.135." You can obtain copies of this document in PDF format; the cost, as of this writing, is $18 and it contains about 600 pages.

The standard, as with most standards, is defined using the Backus Naur Form syntax—illustrated next. The following extract, from ANSI-X3.135-1999, "Database Language SQL," is shown here to illustrate the preciseness with which the standard defines the syntax of a CREATE TABLE statement.

```
<table definition> ::=
CREATE [ { GLOBAL ) LOCAL } TEMPORARY ] TABLE <table name>
<table element list>
[ ON COMMIT { DELETE I PRESERVE } ROWS ]
<table element list> ::=
<left paren> <table element>
[ { <comma> <table element> }... ] <right paren>
<table element> ::=
<column definition>
| <table constraint definition>
```

So, if you are a database vendor and you sell a SQL database product, then you are telling your customers that your database product is capable of parsing and executing ANSI SQL statements, such as the aforementioned CREATE TABLE statement.

There is always an advantage to working with a standard. It means that your knowledge is transferable. An understanding of SQL means that your knowledge of relational database technology can transfer between products such as Oracle and SQL Server. Even within a standard there are differences. SQL is no exception.

SQL differs across vendors for two reasons. First, there are "implementation defined" features in the standard; that is, for some SQL features, the specific nature of how a feature is implemented is "implementation defined"—the standard explicitly states this with the language "implementation defined." This means the vendor has discretion in how the feature is implemented. One example is how the vendor implements the catalog or data dictionary. The standard cannot impose how a vendor stores database metadata. We'll see later in this chapter how Oracle's data dictionary is a set of relational tables from which we can select tremendous amounts of information, such as information describing the tables we create.

A second reason why there are differences across databases is due to the fact that vendors do enhance their specific SQL engines with additional capabilities that they feel benefit the application developer. One example is the DECODE SQL function that you will find in Oracle. This is a powerful function that will not necessarily appear in other database products simply because DECODE is not part of the ANSI standard. Another example is the database column type CURRENCY that you find in SQL Server—not all database products provide a CURRENCY type. These differences are minor compared to the many consistencies across SQL products that are ensured by the ANSI standard.

In summary, you will find variations in some SQL statements across database engines—this is really not a big deal. For example, if you build your expertise in SQL and you build this knowledge in Oracle, then your knowledge will include an understanding of inner and outer joins. If you are ever confronted with the syntax of outer joins from a SQL Server or DB2 application, where this syntax varies, then you can easily make the knowledge transition. The key issue is to recognize that there are differences, because you are likely to become exposed to these differences at some point in your career. When this happens, just remember that products will have subtle differences even under the umbrella of an ANSI standard.

1.2.2 SQL Database

The term *database* is used often in everyday language, but it is important to remember that not all "databases" are created equal. Nor can they all be called relational databases. The term *relational database* refers to one based on the

SQL standard that is capable of parsing and executing SQL statements. We can be very general in how we use the term database, but the term relational database does convey a database that is based on the SQL standard—something very specific.

In contrast to relational databases, there are nonrelational databases, such as DBMS. DBMS is a database product licensed by Oracle and is not a relational database—it is a network database. Access to DBMS is through a language similar to SQL, but it is not SQL per ANSI standard. Some databases, such as IDMS, are not relational databases, yet they provide a front end that allows an application to access the data using SQL. Having a SQL front-end capability does not make IDMS a relational database. A relational database is relational when it conforms to the minimum requirements of the ANSI SQL standard.

Oracle is a fully SQL-compliant database. An Oracle database consists of many files that are preferably distributed across several disk partitions—distributing files minimizes IO contention.

1.2.3 SQL Implementation

Given that Oracle is a relational database and we access Oracle data using SQL, how does one create and manipulate that data? You could write a Visual Basic, C, Perl, or Java program. You could write just about any program in any language you want and embed SQL statements in your program. These statements will insert data, change it, and delete it. Our application programs can potentially execute any valid SQL statement against our relational database.

SQL, originally called SEQUEL, designed by IBM, and based on Codd's relational theory, provided data access methods for the System/R project. The primary purpose of SQL today is that we use it in a manner similar to which it was used by System/R; that is, to be embedded within a programming language and to provide data access methods against database tables for a production system. One core difference, from a day-to-day coding perspective, between programs of today and those of System/R is that we are more likely to use a more current programming language, such as Java or C#. Secondly, programs we code today utilize enhancements to the SQL standards—this includes the key components of this text:

- Database constraints
- Triggers
- Stored procedures

Suppose you just want to do something simple and quick. Let's say you want to create a table with one column and insert one row into that table. For such a simple task, the effort to code and compile a program seems excessive—and it is. For a simple SQL interface we use tools that address the "direct invocation of SQL." The SQL standard addresses embedded SQL—and this is what has been discussed up to this point, embedding SQL statements within compiler language programs—but the SQL standard also addresses another method referred to as Direct SQL. This is where we do the simple and quick stuff.

The SQL Standard specifies three approaches to SQL: embedded SQL, modular SQL, and Direct SQL. Embedded and modular SQL are principal techniques for applications development using a compiler language—they are great for applications development but impractical for simple tasks and for learning the SQL language.

If we did not have an implementation of Direct SQL and wanted to see all the rows of a table, we would have to write an application program, embed a SQL cursor, fetch each row into local variables and print those variables out to some default device. A tool that implements Direct SQL allows us to display the same data in about 5 seconds—mostly limited to how fast we can type or drag-and-drop. Rather than a compiled program, Direct SQL allows us to type a SQL SELECT statement whereby all rows are flushed to our screen for viewing.

The ANSI standard description of direct SQL is stated in the following:

> Direct invocation of SQL is a mechanism for executing direct SQL-statements, known as <direct SQL statement>s. In direct invocation of SQL, the method of invoking <direct SQL statement>s, the method of raising conditions that result from the execution of <direct SQL statement>s, the method of accessing the diagnostics information that results from the execution of <direct SQL statement>s, and the method of returning the results are implementation-defined.

Notice that Direct SQL is "implementation defined," hence vendors have wide discretion in how they implement Direct SQL. SQL*Plus® is an Oracle implementation of Direct SQL, as is SQL Server's Query Analyzer. These tools differ in user interface; however, the same SQL query statement, executed across different tools, will produce the same result set; that is, the same data and number of rows will be the same.

There exists a wide variety of SQL tools that implement direct SQL. These tools allow us to type basic SQL statements and execute them without the additional complexity of a compiler or scripting language. Some are GUI and some provide a basic command line interface. Examples of such tools are SQL*Plus, which is bundled with the Oracle database software; SQL Navigator®, which is licensed by Quest Software, SQL Worksheet which is bundled with the Oracle Enterprise Manager (OEM) software; and TOAD, which is also a Quest Software product. There are many other tools as well.

There is a conceptual difference between accessing Oracle, or any relational database, through a compiler language such as Java and using a direct query tool like TOAD or SQL*Plus. All tools that implement Direct SQL operate within an interactive framework so you need to keep the following two points in mind. First, when you execute a query with SQL*Plus, or with any direct SQL tool, the results are dumped to your screen. Secondly, when executing the same query within an application, you must develop code to capture the results of your query. This is illustrated with the following two SQL query statements.

The first SQL statement is a query that you would execute in SQL*Plus. Upon execution of the statement; the student name is flushed to the screen.

```
1. SELECT student_name FROM registered_students WHERE student_id = 'A101';
```

The aforementioned SELECT statement is a valid SQL statement for Direct SQL only. It will not compile as an embedded SQL statement within a compiled program. The SELECT statement embedded in an application program will be slightly different—the difference is the additional INTO clause, shown next. The INTO clause will reference a program variable into which the student name is to be copied.

[View full width]

```
2. SELECT student_name INTO my_program_variable FROM registered_students WHERE student_id
➥ = 'A101';
```

There are variations on the aforementioned query that use the INTO clause—we have explicit cursors, cursor loops, and other options—these topics are covered in Chapter 11. It is helpful to keep in mind that SQL*Plus, like Toad and other direct SQL tools, is primarily an interactive tool—you do not worry about capturing result data. On the other hand, application programs do capture query result data and when you develop code to "capture data" you have design issues: multiple rows returned exception conditions—these topics are addressed throughout the text.

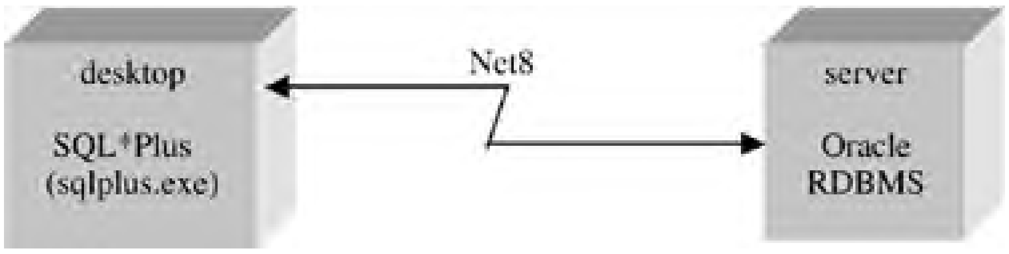
1.2.4 SQL*Plus

In addition to being an implementation of Direct SQL, SQL*Plus is a trademarked name for an Oracle executable program that runs as a command line interactive program. As mentioned in Section 1.2.3, there are many SQL tools one can use to execute SQL against a database. We've mentioned SQL*Plus, Toad, SQL Navigator, and SQL Worksheet. In fact, you can use SQL Server SQL Query Analyzer to run your queries against your Oracle database. You have many choices. This text makes use of SQL*Plus, but you'll have no difficulty following along should you use another tool for interacting with Oracle.

SQL*Plus primarily serves an end user in a client/server environment. Figure 1-1 illustrates a common configuration that includes SQL*Plus. In this figure, the communication protocol across this network is the Oracle Net8 protocol. The Oracle software that runs on your client sends the SQL statement across the network to the database and it is Oracle software, on the back end, that forwards the data back to the client. If you run SQL*Plus and type the following SQL statement:

```
SELECT student_name FROM registered_students WHERE student_id = 'A101';
```

Figure 1-1. SQL*Plus in a Client/Server Mode.



the SQL statement, exactly as you type it, including the same case, is sent across in a network packet to the Oracle server. The processed result, that being a list of student names (in this case probably just the one name of the student with that particular student ID), is returned to the SQL*Plus client where it is flushed to your screen.

The sending and receiving of data is Oracle Net8, which is Oracle communication software and runs over TCP/IP, or other network protocols. SQL*Plus is written to utilize Net8, which utilizes TCP/IP over the network.

SELECT, INSERT, UPDATE, and DELETE are the most common SQL statements—they are the core means of manipulating data. In addition to these core data manipulation statements, the SQL standard includes the specification for statements like: CREATE DATABASE, ALTER DATABASE, CREATE TABLE, CREATE VIEW, CREATE TRIGGER, and many other SQL commands. Any valid SQL statement can be executed from SQL*Plus, including the ALTER DATABASE statement. So, all SQL statements, not just the common INSERT, UPDATE, DELETE, SELECT can be executed from SQL*Plus.

There are many advantages to learning SQL*Plus. A few are summarized here.

- You will find SQL*Plus in the Oracle bin directory of every Oracle install. If you install the database on an enterprise server, you will find SQL*Plus in the bin directory. If you install just the Oracle developer software on the desktop, you also have SQL*Plus. This executable program is always there. So, in the absence of having any other tools, you can always be sure you have the SQL*Plus program.

- Because SQL*Plus has been available from Oracle through so many versions, the program has an extremely wide user base. It is popular for several reasons, one of which is the fact that it has been around for so many years.

- SQL*Plus is a simple command line interface program. It's not a GUI interface; you do not have multiple scrollable windows that show the SQL statement in one window and the query result in another window. Although GUI features have their advantages, the simplicity of SQL*Plus becomes advantageous when you need to build scripts. You can build a library of SQL*Plus scripts, each an ASCII text file, and those scripts will run on Oracle in UNIX or on Oracle in Windows.

- SQL*Plus has all the features of a powerful scripting language. It supports argument passing, command files, and nested command files. You can host out to other languages (e.g., embed a Windows script host file or Korn shell script within a SQL*Plus script). SQL*Plus scripts can be incorporated into other scripting programs (i.e., SQL*Plus scripts can be embedded in Korn shell scripts; illustrated in Chapter 2). These features make the language highly useful for tool building.

- In summary, although SQL*Plus is an excellent tool to first experience SQL and Oracle, it is also widely used as a development and administration tool. People who use SQL*Plus initially use it to introduce themselves to Oracle, but years later they're still using it for applications development and database administration.

1.3 Tables

Database tables are the most fundamental structure in a database. If you were asked to work on a new Oracle database, you would initially wonder how many tables there are in the application. It makes a difference whether there are six tables or 600 tables, at least in terms of how much there is for you to learn about the application before you can be a productive developer.

The examples in this text are based on a demo student data model. The model stores data for students, courses, and professors. Chapter 4 graphically illustrates this model including the SQL create scripts. A key table in this model is the STUDENTS table.

An initial plan for a STUDENTS table requires answering the question: "What attributes should be stored for each student?" An initial draft of a STUDENTS table includes these attributes:

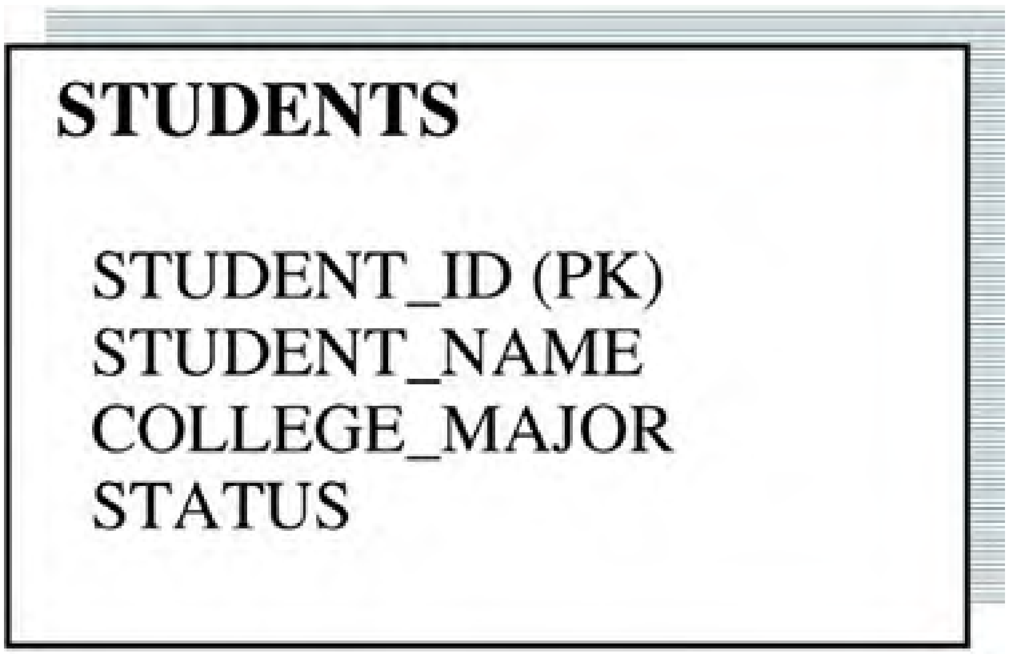
Attribute	Column Name
A unique student number. Each student is assigned a number used to access billing and registration records.	STUDENT_ID
A student name.	STUDENT_NAME
A major field of study such as Science or History.	COLLEGE_MAJOR
A college enrollment status indicating a degree-seeking or certificate-seeking student.	STATUS

Tables have different representations. Originally, the table is a logical entity consisting of a box on an entity relationship diagram. In production, the table can be described with the SQL*Plus Describe command and rows can be selected.

1.3.1 Data Model View

The data model view of a table identifies the logical attributes of the entity being stored in the database (Figure 1-2). This includes a key attribute, which is the primary key. The primary key is unique for each instance of the entity. If the table stores students, every student will have a unique primary key. The data model view is a graphical representation.

Figure 1-2. Students Table in a Data Model.



1.3.2 Create Table Script

Once the model is complete and all attributes are defined, the entity must be created in the database. SQL is used to create a table. This step transitions the logical concept to a physical table in the database. When SQL creates objects, such as tables, this is called Data Definition Language (DDL). The following DDL creates a STUDENTS table and defines the column type of each attribute to be a variable length string with a maximum size.

```
CREATE TABLE students
(student_id      VARCHAR2(10),
 student_name    VARCHAR2(30),
 college_major   VARCHAR2(15),
 status          VARCHAR2(15));
```

1.3.3 Describing the Table

The definition of a table can easily be described with the SQL*Plus command. This command retrieves information from the data dictionary in a format that conveys the information stored in the table. If the DDL in the previous section were used to create the STUDENTS table, the SQL*Plus describe command would return the following:

```
SQL> desc students
 Name                          Null?    Type
 ----------------------------- -------- ---------------
 STUDENT_ID                    NOT NULL VARCHAR2(10)
 STUDENT_NAME                  NOT NULL VARCHAR2(30)
 COLLEGE_MAJOR                 NOT NULL VARCHAR2(15)
 STATUS                        NOT NULL VARCHAR2(15)
```

1.3.4 Table Data

SQL statements that manipulate rows in the table are called Data Manipulation Language (DML). Once the table is created, we can add students to the system with INSERT statements.

```
INSERT INTO students
VALUES('A101','John','Biology','Degree');
```

A key component of relational technology, emphasized in Section 1.1, is the freedom of the programmer to choose what data to query. Once the table is created and rows are inserted, there is no restriction with data access. We can query all students within a particular major. We can query all students who are degree candidates. We can query all degree-seeking students majoring in either biology or history.

```
SELECT student_name FROM students WHERE college_major = 'Biology';

SELECT student_name FROM students WHERE status = 'Degree';

SELECT student_name FROM students WHERE
status = 'Degree' AND (college_major='Biology' OR college_major='History');
```

In network and hierarchical database technologies, the programmer cannot, at the last minute, decide to query data in a particular order. In network and hierarchical databases, ordered query results must be built into the database structure. In relational databases we can, as developers, choose to select students and their major in alphabetical order, reverse alphabetical order, order by name, and order by major—we have no restriction. The ordering of data is not in the database but within the SQL statement. So to pull all student names and their major in alphabetical order, we use the ORDER BY clause, which is a component of the SQL specification. Showing this ordered list, we use:

```
SELECT student_name, major FROM student ORDER BY student_name;
```

All rows of a table can be selected with the SELECT * syntax. The following is the SQL*Plus session output that selects all rows from this STUDENTS table after five students have been added.

```
SQL> SELECT * FROM students;

STUDENT_ID STUDENT_NAME    COLLEGE_MAJOR  STATUS
---------- --------------- -------------- -----------
A101       John            Biology        Degree
A102       Mary            Math/Science   Degree
```

A103	Kathryn	History	Degree
A104	Steven	Biology	Degree
A105	William	English	Degree

1.4 SQL Statements

The SQL language is primarily used to create tables and manipulate table data; however, the language includes statements that perform other functions. SQL statements are used to create database accounts, size and tune for performance, and perform administration tasks. An Oracle database first comes into existence with a SQL statement, the CREATE DATABASE statement. The following describes the categories of SQL statements.

1.4.1 Data Definition Language (DDL)

CREATE TABLE is a DDL statement. DDL statements define our database objects and result in updates to the Oracle data dictionary. They create, modify, and delete objects such as a tables, views, stored procedures, database triggers, database links, and dozens of other objects in the database.

DDL statements can alter existing database objects including tables. Columns and constraints can be added with the ALTER TABLE command. Should we choose to add a column for a student's age we can execute the following SQL statement, which alters the definition of the table in the database data dictionary.

```
ALTER TABLE students ADD (age NUMBER(3));
```

1.4.2 Data Manipulation Language (DML)

DML manipulates data with the following statements:

- INSERT
- UPDATE
- DELETE
- SELECT

SELECT statements do not actually manipulate data, but the SELECT statement is often included within the conversational context of "DML."

An Oracle database provides table changes through INSERT, UPDATE, and DELETE to cohabitate with SELECT statements. The concept supported in an Oracle environment is called consistent read image. A user making uncommitted changes to the database through INSERT, UPDATE, and DELETE statements does not block a user who issues a SELECT during that same time frame. Transactions such as an INSERT that are not committed to the database do not interfere with concurrent SELECT statements against the same tables. A SELECT statement is guaranteed a consistent image. The results returned to the user from a SELECT statement are guaranteed to be from the most recent consistent image, which will be the rows as of the most recent COMMIT. A COMMIT statement establishes a new checkpoint by which table data is updated to a consistent point in time.

1.4.3 Transaction Control

Transaction control statements allow you to bundle a group of DML statements under an all-or-nothing domain. That is, "all statements complete successfully," or if one statement fails, then all statements fail as one group. Examples of transaction control statements are: SET TRANSACTION, COMMIT, and ROLLBACK.

1.4.4 Session Control

Session Control statements are temporary and persist for the duration of the user's database connection. A useful SQL tuning practice is to turn SQL trace on for the session. This records session information to a user dump directory for SQL statement analysis using the Oracle tool, TKPROF. The following will turn tracing on/off for the session.

```
ALTER SESSION SET SQLTRACE [TRUE|FALSE]
```

Once SQL Trace is on, the user runs the application. During execution Oracle writes data to a trace file. The trace file can be analyzed using the Oracle utility, TKPROF. The following shows the host command string to parse the trace file. The result after running TKPROF is an output ASCII file showing each SQL statement run with detail tuning information.

```
tkprof oracle_trace_file.trc oracle_output_trace_file.prf
explain=SCOTT/TIGER@ORA sys=no
```

The following is a common alter session command. It alters the default display format for columns with a DATE type to include: day, month, year, hours, minutes, and seconds.

```
ALTER SESSION SET NLS_DATE_FORMAT='dd-mon-yyyy hh24:mi:ss';
```

1.4.5 System Control

System Control statements are largely database administrative commands. The SQL statements are used to open the database, shut down the database, and perform administrative commands such as resize datafiles or switch log files. The following is a system control statement used to set the number of Oracle job queue processes.

```
ALTER SYSTEM SET job_queue_processes=4 SCOPE=BOTH;
```

1.5 Table Column Datatypes

The most common datatypes are VARCHAR2, NUMBER, and DATE. These types handle basic column types for character strings, numbers, and columns with a time dimension. The examples in this text, including the data model in Chapter 4, use these three types. Oracle provides many other datatypes with powerful features.

Some datatypes are supplemented with built-in functions and packages. Built-in functions are available to convert a DATE between time zones. Large objects, such as CLOB and BLOB types, are supported with a PL/SQL built-in package, DBMS_LOB which enables the manipulation of 4-gigabyte objects. The following lists the datatypes available for column definitions in a table.

1.5.1 Character

```
VARCHAR2(n)
```

The VARCHAR2 is the general string type used to store strings of up to 4,000 characters. A field of this type is, internally, a variable length string. It only uses the minimum necessary bytes—no padding. The parameter *n* is required. For the column STUDENT_NAME, in the STUDENTS table, we can store a name up to 30 characters.

Oracle version 6 and earlier versions used a VARCHAR that was a fixed length character field. Oracle introduced this variable length and more space proficient datatype in version 7 and called it VARCHAR2. VARCHAR2 is the basic string type for an Oracle table column.

```
CHAR(n)
```

This type is used to store a fixed-length string. Inserts will right-pad blanks to the value if the length of the inserted value is less than the length of the CHAR declaration—this impacts programmers writing SQL with WHERE clauses who do not expect column values to be padded. This datatype is rare.

```
NCHAR(n), NVARCHAR2(n)
```

These types are used only to store Unicode (unified encoding) character data, which is applicable for Oracle applications that support multiple languages.

1.5.2 Number

```
NUMBER, NUMBER(a), NUMBER(a,b)
```

This is a column type used for storing numbers. Do not use precision parameters for storing values of unknown range. For the following, we can store a number with virtually any magnitude in the column AMOUNT.

```
CREATE TABLE TEMP (AMOUNT NUMBER);
INSERT INTO TEMP VALUES (0.12345);
INSERT INTO TEMP VALUES (12345.1209);
```

The parameter "a" represents precision, the total number of placeholders. The parameter "b" represents scale, places to the right of the decimal point. So if your field's domain is the age of a person then you would use the following to store any three-digit whole number:

```
CREATE TABLE TEMP (age NUMBER(3));
```

Should you be storing stock prices with a maximum of a million dollars (seven places to the left of the decimal point) and four places to the right of the decimal point, you would use:

```
CREATE TABLE TEMP (PRICE NUMBER(11,4));
```

Scale is optional and, if not used, the values are stored as whole numbers. Fractions, when inserted, are rounded to the nearest whole number. To specify a scale only, use:

```
CREATE TABLE TEMP (PRICE NUMBER(*,4));
```

1.5.3 DATE Types

```
DATE
```

The DATE type stores date and time—some databases separate this into two types, a type for the date and a type for a time. For Oracle, the DATE type is a calendar date and time—all in one. In some applications the time of day is irrelevant—just the date, hence an application might choose to not show the time but it is always there. There are numerous built-in functions on date types that PL/SQL can perform, such as computations to determine the last day of the previous month. DATE arithmetic is also supported. You can easily build on existing date functions and create a PL/SQL library of date operations tailored to your application.

The default display of a DATE column is DD-MON-YY. This default display can be changed with an ALTER SESSION command. To display DATE values showing hours, minutes, seconds:

```
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYYhh24:MI:SS';
```

```
TIMESTAMP, TIMESTAMP(a)
```

The TIMESTAMP data type is derived from the DATA type. It represents time with the same precision as DATE including a fractional second's field. The TIMESTAMP format is:

```
28-JUL-03 01.25.56.000122 PM
```

TIMESTAMP can be declared with an optional parameter. The parameter "a" represents the number of digits included in the fractional part of the timestamp.

```
TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE
INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH
```

TIMESTAMPS can incorporate time zones. Applications spread over multiple time zones are likely to use date columns of this type. The INTERVAL type represents a time difference to a particular significance. The following creates a table with columns of each type.

```
CREATE TABLE times
 (d1 TIMESTAMP WITH TIME ZONE,
  d2 TIMESTAMP WITH LOCAL TIME ZONE,
  d3 INTERVAL DAY TO SECOND,
  d4 INTERVAL YEAR TO MONTH);
```

1.5.4 Large Objects

CLOB, BLOB, NCLOB

Use a CLOB to store large-character objects. The limit is four gigabytes. A VARCHAR2 is limited to 4,000 characters. Character-based columns needing more than four gigabytes should be CLOBs.

Use the BLOB to store four gigabytes of binary data.

A student writing sample can be a column in the student table. If writing samples are documents that exceed 4,000 characters they are CLOB-type columns. The table and column are created with the following:

```
CREATE TABLE student_writting_samples
(student_id     VARCHAR2(10),
 writing_sample CLOB));
```

Manipulation of the CLOB is through a CLOB/BLOB API. This API is the Oracle built-in package DBMS_LOB. An application manipulates the object by first selecting the CLOB into a PL/SQL object. The CLOB/BLOB is a private type. Operations are restricted to the API. The following PL/SQL script prints the length of the student writing sample.

```
DECLARE
    v_writing_sample CLOB;
    the_length       INTEGER;
BEGIN
    SELECT writing_sample
      INTO v_writing_sample
      FROM student_writing_samples
     WHERE student_id = 'A101';

    dbms_lob.open(v_writing_sample, dbms_lob.lob_readonly);
    the_length := dbms_lob.getlength(v_writing_sample);
    dbms_lob.close(v_writing_sample);
    dbms_output.put_line(the_length);
END;
```

BLOB is a four-gigabyte object that is intended for storing binary images. BLOB objects are manipulated using the built-in package DBMS_LOB.

NCLOB types store four-gigabyte Unicode character objects using the National Character set.

BFILE

An object need not physically be stored in a column. Rather, a column can act as a reference to an object. Such an object would be a file on the host disk. A student writing sample can be stored completely in a column as a CLOB. Alternatively, a writing sample can remain on a file server, yet be accessible through SQL. This requires that there be a BFILE column type. The BFILE acts as a reference to the external file. The external file content can be accessed using the DBMS_LOB API.

The following creates a table for student writing samples and inserts an external file reference. The BFILE column is just a pointer to the file JOHN.DOC that physically resides in the directory D:\STUDENT_FILES. The creation of the Oracle directory requires the CREATE ANY DIRECTORY privilege.

```
CREATE OR REPLACE DIRECTORY
STUDENT_FILES AS 'D:\student_files';

CREATE TABLE student_writing_samples
(student_id     VARCHAR2(10),
 writing_sample BFILE);

INSERT INTO student_writing_samples
    (student_id, writing_sample)
VALUES
    ('A101',bfilename('STUDENT_FILES','John.doc'));
```

BFILE objects are manipulated using the DNMS_LOB API. To print the length of the writing sample:

```
DECLARE
    v_writing_sample BFILE;
    the_length       INTEGER;
BEGIN
    SELECT writing_sample
    INTO   v_writing_sample
    FROM   student_writing_samples
    WHERE  student_id = 'A101';

    dbms_lob.fileopen
        (file_loc => v_writing_sample,
         open_mode => dbms_lob.file_readonly);

    the_length := dbms_lob.getlength(v_writing_sample);

    dbms_lob.fileclose(file_loc => v_writing_sample);

    dbms_output.put_line(the_length);
END;
```

1.5.5 XML Type

XMLType

The XMLTYPE was released with Oracle 9*i* Release 2. This type is used to store XML documents as attributes to an entity. For example, student transcripts can be stored as XML documents. This document would be an attribute of the student and exist in a STUDENTS table as a single column. The table for a student and transcript is:

```
CREATE TABLE student_transcripts
(student_id VARCHAR2(10),
 transcript SYS.XMLTYPE);
```

The XMLType has built-in methods. The following statement inserts a transcript for a student with two classes.

```
INSERT INTO student_transcripts
    (student_id, transcript)
VALUES ('A101',
        sys.XMLTYPE.createXML('<STUDENT_TRANSCRIPT>'
        ||'<CLASS>'
        ||'<COURSE>Math 101</COURSE>'
        ||'<GRADE>A</GRADE>'
        ||'</CLASS>'
        ||'<CLASS>'
        ||'<COURSE>English 102</COURSE>'
        ||'<GRADE>A</GRADE>'
        ||'</CLASS>'
        ||'</STUDENT_TRANSCRIPT>'));
```

The following SELECT statement returns the student and transcript record.

```
set long 10000
SELECT
    student_id,
    p.transcript.getClobVal()
FROM
    student_transcripts p;
```

XML documents can be read from a host file and loaded into a CLOB-type column. It can then be navigated in PL/SQL using the XML built-in packages that fully support the Document Object Model (DOM) API for XML documents.

1.5.6 LONG and RAW Types

```
LONG
```

Oracle introduced support for large objects in version 8. Prior to this release, strings that exceeded the bounds of a VARCHAR2 type could be manipulated, with limited functionality, using a LONG datatype. Strings that exceed the limits of a VARCHAR2 should now be manipulated as a CLOB using the built-in DBMS_LOB package.

```
RAW, LONG RAW
```

Similar to the LONG type, the RAW and LONG RAW datatypes served a useful purpose prior to LOB support with Oracle 8. The RAW and LONG RAW types were used for manipulating binary data—the Oracle built-in package, DBMS_LOB, provides advanced support for manipulating binary data.

1.5.7 ROWID Type

```
ROWID, UROWID
```

A ROWID contains composite information that allows Oracle to identify, form a single ROWID value, where a specific row in a table is located (i.e., the tablespace, file, database block, and specific location within that block). You can declare a table with a column of type ROWID, populate that table with ROWID values, and extract information about the value using the Oracle built-in package DBMS_ROWID. The UROWID is a universal ROWID structure that supports non-Oracle tables.

1.6 Behind Tables

What is a table? Is it a file, a block, or a stream of bytes? Here we look at tables logically and physically.

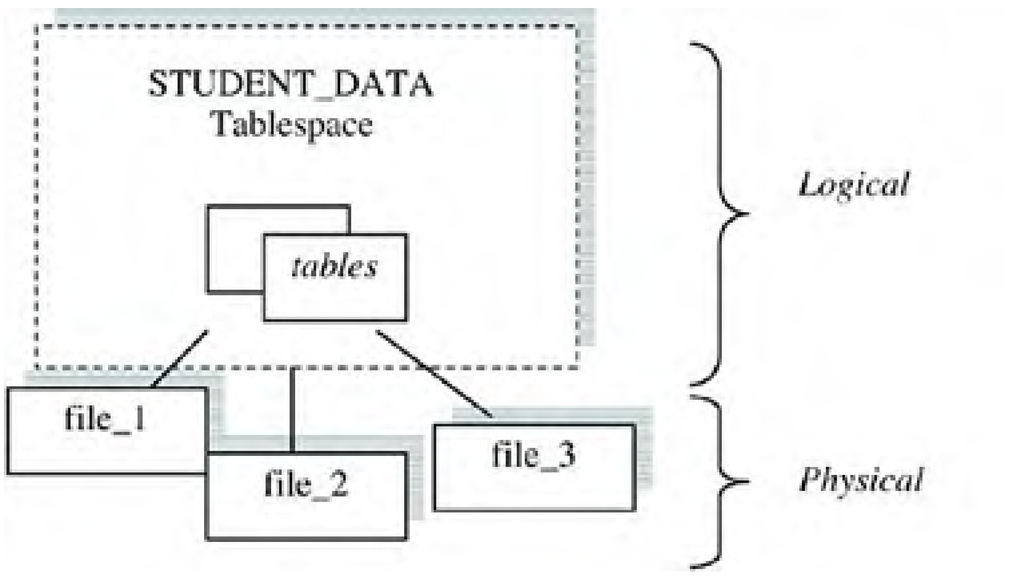
1.6.1 Application Tablespaces

All table data is ultimately stored in host operating system files; but, the insertion of rows never specifically identifies a host file.

The first step is to create an intermediate logical layer called a tablespace with a CREATE TABLESPACE statement. This statement includes the host pathnames of one or more host files that are to be created. The CREATE TABLESPACE statement creates the files mentioned in the statement, formats the files, and stores information in the Oracle data dictionary. The data dictionary information tracks the fact that a tablespace is made up of specific files.

Once the tablespace is created, the CREATE TABLE statement can reference the tablespace name in the create statement. From this point on, Oracle will use the files of that tablespace for row storage. Figure 1-3 illustrates this architecture showing that tables and tablespaces are logical entities whereas the datafiles are the ultimate physical component.

Figure 1-3. Tables in a Tablespace.



To replicate the environment in Figure 1-3 create the tablespace, then the table. The following creates a tablespace STUDENT_DATA and allocates 10M of disk space. The presumption is that this file does not exist; in fact, this statement will fail immediately if the file exists prior to statement execution.

```
SQL> CREATE TABLESPACE student_data DATAFILE
  2 'D:\student_data.dbf' size 10M;

Tablespace created.
```

To create a STUDENTS table in the STUDENT_DATA tablespace:

```
SQL> CREATE TABLE students
  2  (student_id    VARCHAR2(10),
  3   student_name  VARCHAR2(30),
  4   college_major VARCHAR2(15),
  5   status        VARCHAR2(20)) TABLESPACE student_data;

Table created.
```

Other tables can be added to the STUDENT_DATA tablespace. The student demo is described in Chapter 4. All the demo tables are created in a STUDENT_DATA tablespace.

A single application usually has all tables in one tablespace. There are circumstances where multiple tablespaces are used. Multiple tablespaces are driven by a variety of issues including highly demanding physical storage requirements and partitioning. The following summarizes some remaining topics on tablespaces.

- There is a standard, known as the Optimal Flexible Architecture (OFA). The OFA standard recommends that database files fit into a directory structure where the parent directory name is the same name as the database name, plus other reasonable considerations. The aforementioned example violates this convention only to simplify the example.
- The datafile D:\student_data.dbf did not exist prior to the CREATE TABLESPACE statement. This file is created during the execution of the CREATE TABLESPACE statement. It is possible to create a tablespace on an existing datafile—this requires a REUSE clause in the syntax.
- A tablespace can consist of multiple files. For example, if you need 20M you can have two 10M files.
- The datafiles in the CREATE TABLESPACE statement are formatted by Oracle. You'll notice that a CREATE TABLESPACE statement on a 2G datafile takes relatively longer that a 2M datafile. This is because Oracle formats the datafile using its own internal block structure.
- The aforementioned example is simple and may imply a strict architecture—such as dealing with space when you fill up 10M of data. The tablespace model is highly flexible. You can add files to an existing tablespace, resize a datafile, move a datafile to another drive and resize it, or allow datafiles to auto-extend—all without taking down the database. The physical layout of an Oracle database is highly flexible.
- A datafile can serve one and only one tablespace. You will never, and cannot possibly, have conditions where a datafile is "tied" to more than one tablespace.
- An Oracle user always has a DEFAULT tablespace. So, if you do not specify a tablespace name, that table is created in your default tablespace. You can get your default tablespace name by querying the data dictionary view USER_USERS.

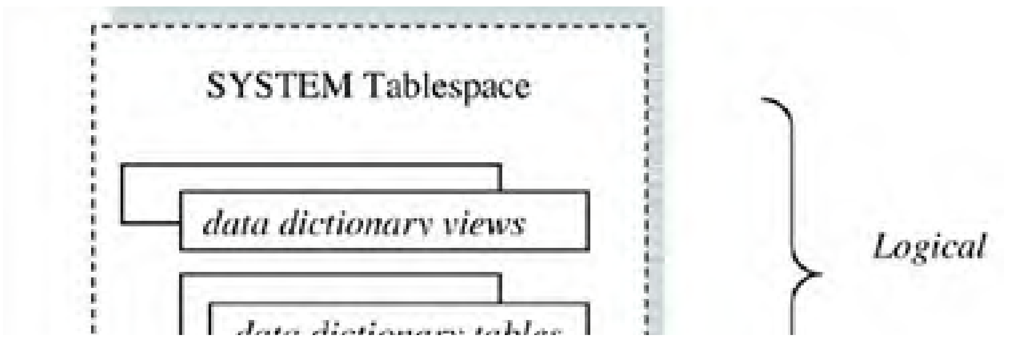
```
SQL> SELECT default_tablespace FROM user_users;

DEFAULT_TABLESPACE
------------------------------
USERS
```

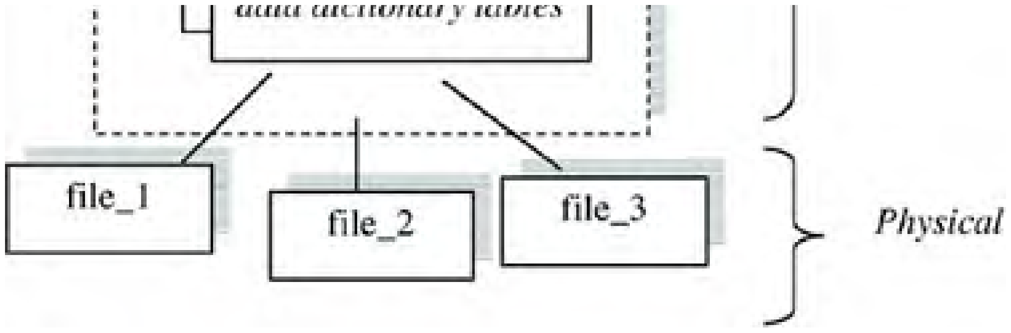
- A table is created in a single tablespace. Exceptions to this are partitioned tables where individual partitions are created in separate tablespaces.
- While a table is created in one tablespace, the indexes for that table are often in a separate tablespace. The DDL for the data model demo in Chapter 4 creates all indexes in the tablespace STUDENT_INDEX.

1.6.2 Data Dictionary

The execution of a CREATE TABLE statement causes information to be stored in the data dictionary. The data dictionary is the term used to describe tables and views that exist in the SYSTEM tablespace. The data dictionary is essentially a repository for Oracle to track information about all objects created in the database. The information tracked includes: the table name, who owns the table, when it was created, column names and datatypes, and the tablespace name to which a table belongs. All PL/SQL stored procedure source and compiled code is stored in the data dictionary. The data dictionary tables and views of the SYSTEM tablespace are illustrated in Figure 1-4.

Figure 1-4. Data Dictionary and System Tablespace.





The data dictionary consists of Oracle tables and views that are constructed from SELECT statements against the base tables. The data dictionary views provide the attributes of any object created. The view USER_TAB_COLUMNS can be queried to determine the column names of a table. The data dictionary view to query for student column definitions is USER_TAB_COLUMNS.

The SYSTEM tablespace is created when the database is first created. The SYSTEM tablespace and datafiles are generated as part of the CREATE DATABASE statement. Application tablespaces, such as STUDENT_DATA, can be added to the database at any time.

The following SQL*Plus session creates a STUDENTS table. A query of the data dictionary view USER_TAB_COLUMNS shows the column name and column type of all columns in the STUDENTS table.

```
SQL> CREATE TABLE students
  2  (student_id VARCHAR2(10),
  3   student_name VARCHAR2(30),
  4   college_major VARCHAR2(15),
  5   status VARCHAR2(20)) TABLESPACE student_Data;

Table created.

SQL> SELECT table_name, column_name, data_type
  2  FROM user_tab_columns
  3  WHERE table_name='STUDENTS';

TABLE_NAME      COLUMN_NAME                    DATA_TYPE
---------------- ------------------------------ ----------
STUDENTS        STUDENT_ID                     VARCHAR2
STUDENTS        STUDENT_NAME                   VARCHAR2
STUDENTS        COLLEGE_MAJOR                  VARCHAR2
STUDENTS        STATUS                         VARCHAR2
```

To see the tablespace in which the STUDENTS table exists, use

```
SQL> SELECT tablespace_name
  2  FROM user_tables
  3  WHERE table_name='STUDENTS';

TABLESPACE_NAME
------------------------------
STUDENT_DATA
```

The following shows the datafiles and file sizes associated with the STUDENT_DATA tablespace. This query selects from the DBA_DATA_FILES view and requires that you have the DBA role or SELECT_CATALOG_ROLE role.

```
column file_name format a50
SQL> SQL> SELECT file_name, bytes
  2  FROM dba_data_files
  3  WHERE tablespace_name='STUDENT_DATA';

FILE_NAME                                    BYTES
-------------------------------------------- --------
E:\ORACLE\ORADATA\ORA10\STUDENT_DATA01.DBF    5242880
```

Chapter Two. Interacting with Oracle

The purpose of this chapter is twofold: demonstrate the interactive features of SQL*Plus and, show how to incorporate SQL*Plus into other scripting languages such as Korn Shell and Perl.

2.1 Simplify SQL*Plus for Yourself on Windows

In this section we focus on methods that simplify SQL*Plus script file organization. You will eventually be creating, saving, and retrieving SQL*Plus scripts. You want the scripts to be separate from other files; you want to know exactly where they are and build and modify them without a lot of headaches.

SQL*Plus is a scripting language that includes command file support. This text encourages the use of command files, also called SQL scripts. Your SQL*Plus scripts will eventually contain multiple SQL DML statements, CREATE table statements with constraint clauses, and Oracle stored procedures and triggers written in PL/SQL.

When you install Oracle, whether it's the full enterprise licensed database or just the desktop client tools, you will be able to launch SQL*Plus from your desktop through START -> PROGRAMS. When you use this method to run SQL*Plus, your default directory (i.e., the directory for saved and retrieved scripts) is the same as the Oracle "bin" directory. This is not very convenient—you want to keep your scripts in a separate working directory. You can override the default "bin" directory and save a SQL script to another directory—this requires including the full pathname with the SQL*Plus SAVE command. You can run SQL scripts that exist in a nondefault directory if you type a full pathname in the run command. These override methods are inconvenient, and there are better techniques to be discussed shortly.

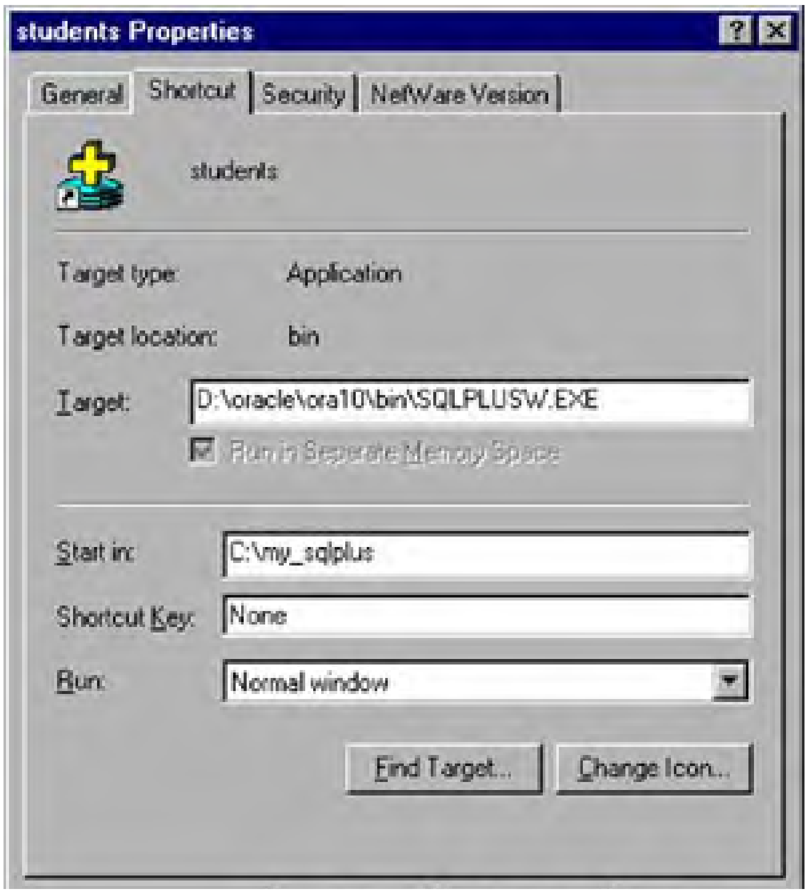
One approach to overriding the default directory is to create a desktop shortcut for SQL*Plus and designate a START-IN directory. The START-IN directory will be the new default directory for SQL*Plus scripts. Without specifying a start-in directory, all saved scripts default to the Oracle "bin" directory—a directory with hundreds of other Oracle files.

Let's do this now. Using Windows Explorer, create a "working directory" where you wish to store your library of SQL*Plus scripts. For example, create the directory:

```
C:\MY_SQLPLUS
```

Next, create a desktop shortcut for SQLPLUSW.EXE. This program is in the Oracle "bin" directory. Then right-click the newly created shortcut icon from the desktop, select "Properties," and select "Short cut"—this is where you set the "Start in" directory. Set the "Start in" field to the path of the directory you just created, (i.e., C:\MY_SQLPLUS). The shortcut creation step is shown in Figure 2-1.

Figure 2-1. SQL*Plus Shortcut.



students

Target type: Application

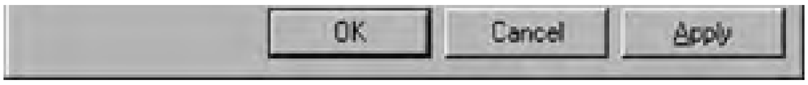
Target location: bin

Target: D:\oracle\ora10\bin\SQLPLUSW.EXE

Start in: C:\my_sqlplus

Shortcut Key: None

Run: Normal window



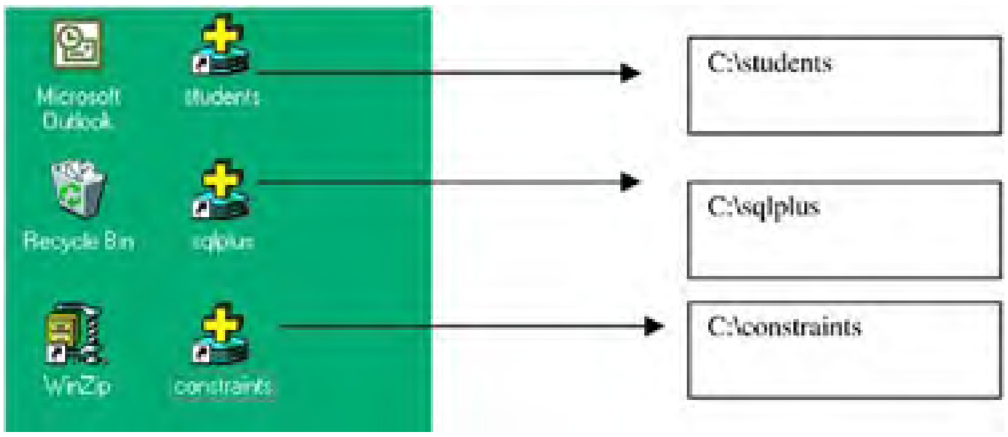
After you type the pathname in the "Start in" field, select the "Apply" button, then the "OK" button, and you're done. By doing this you are able to save and retrieve scripts into, and from, a dedicated "Start in" directory—nothing else but your scripts exists here. You can go to this directory to copy and edit scripts. You have now made the directory C:\MY_SQLPLUS, your personal development code library, and integrated it into SQL*Plus.

The aforementioned process is not necessary in a UNIX environment. In UNIX you can "cd" (change directory command) to any UNIX directory, such as a dedicated SQL*Plus code library, and invoke SQL*Plus by typing just the executable name on the command line—this works because the SQL*Plus program is in your $PATH. From this point, all files from your SQL*Plus session are automatically saved into the current working directory; that is, the directory from which you invoke SQL*Plus.

You will see two versions of SQL*Plus in your Windows Oracle "bin" directory: SQLPLUS.EXE and SQLPLUSW.EXE. Use SQLPLUSW—this is for everyday interactive use. It begins with a popup box prompting you with login information. This version runs SQL*Plus in a scrollable window—this is very easy to work with. The SQLPLUS.EXE version runs in a "DOS" window, black screen, and no scrollable window. You would use this version of SQL*Plus to launch a SQL script from a DOS batch file or call SQL*Plus with command line arguments from a scripting languages such as Perl—illustrated in Section 2.12.

After you create the desktop shortcut and update the start-in directory, you have a desktop with the SQL*Plus icon showing. Figure 2-2 shows a desktop with not one, but three, SQL*Plus icons. Here, each icon is a shortcut to the same SQL*Plus executable; however, each has a different start-in directory, which means that each has a separate "working directory." This is one technique for "conquering and dividing" your work in a Windows environment. Again, this is not necessary in a UNIX environment because you simply "cd" to any working directory and once in that directory, you invoke SQL*Plus. In a Windows environment, you may be inclined to separate work among various directories and tie a SQL*Plus executable shortcut to these separate work directories.

Figure 2-2. SQL*Plus Shortcuts on the Desktop.



The three boxes on the far right of Figure 2-2 represent distinct "Start in" directories. One directory might be used to store all the CREATE table scripts for your application. This would include scripts to create triggers, sequences, and indexes. One could be the directory where you maintain all your PL/SQL stored procedures. One directory would be used for application statement tuning—this directory would include many SQL*Plus list files with the execution paths of the many SQL statements you are tuning.

2.2 Connecting

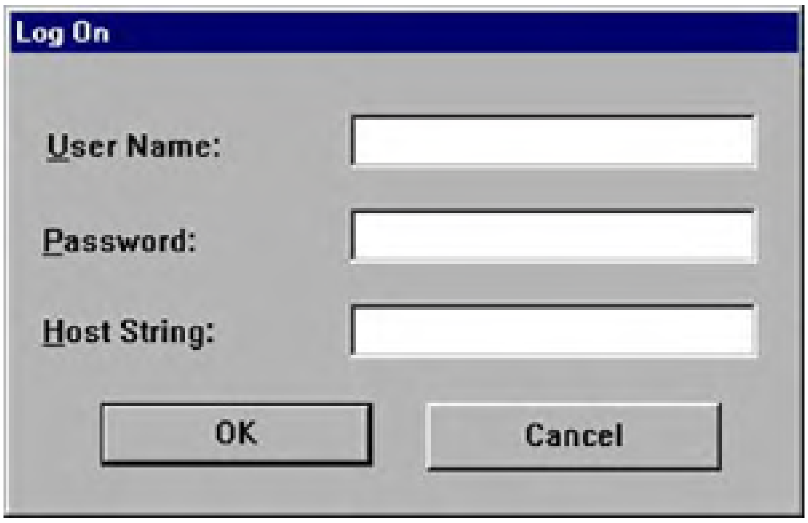
To proceed from this point forward, you need an Oracle database. There are two basic configurations. One software configuration is to have everything on one machine; that is, the Oracle database and client code can all be on your desktop. An alternative is for your Oracle database to be on a separate host, in which case you're connecting with SQL*Plus to Oracle over a network.

If your Oracle software environment is set up, you're ready to connect and begin executing SQL. If this needs to be done you can begin with installing the Oracle Client CD. This installs SQL*Plus and other utilities on your desktop that allow you to interact with a database on the network. An alternate is to install the Oracle Enterprise CD. This gives you the enterprise software and optionally creates a default database.

For Windows environments, it is strongly recommended that you set up a shortcut and a start-in directory, described in Section 2.1. Once this is done, you can invoke SQL*Plus from the desktop.

When you launch SQL*Plus from Windows, you get a popup window with input fields for a username, password, and host string—illustrated in Figure 2-3.

Figure 2-3. SQL*Plus Login.



Once you enter the username, password, and host string you have a permanent point-to-point connection to the database. Once connected, the database is at your service to process everything from database queries, to CREATE table statements, and stored procedure execution.

For UNIX environments, the SQL*Plus program directory, $ORACLE_HOME/bin, must be in your path. In UNIX, you invoke SQL*Plus from the command line by typing the program name followed by an argument string that contains username, password, and host string in the format shown next:

```
$ sqlplus username/password@host_string
```

For example:

```
$ sqlplus scott/tiger@ora
SQL> _
```

2.3 Connecting to an Infrastructure

This section illustrates the Oracle infrastructure to which you connect. The first part, illustrated in Figure 2-4, discusses the point-to-point connection you have with Oracle in a dedicated-server environment. Second, we discuss the instance infrastructure, illustrated in Figure 2-5. This is all tied together with a discussion surrounding Figure 2-6.

Figure 2-4. Oracle Dedicated-Server Mode.

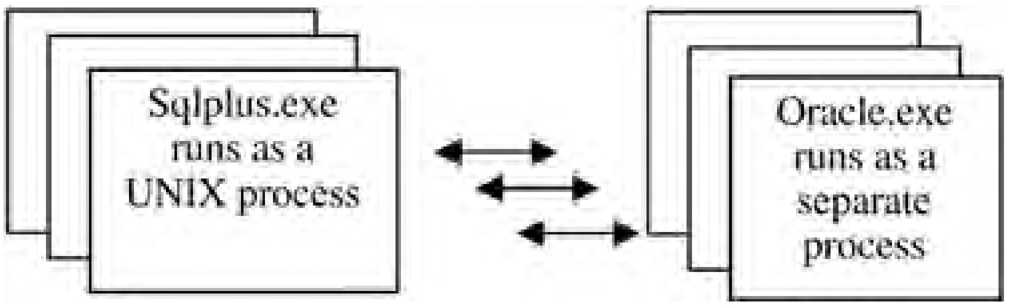

Sqlplus.exe runs as a UNIX process
Oracle.exe runs as a separate process

Figure 2-5. Database Instance.

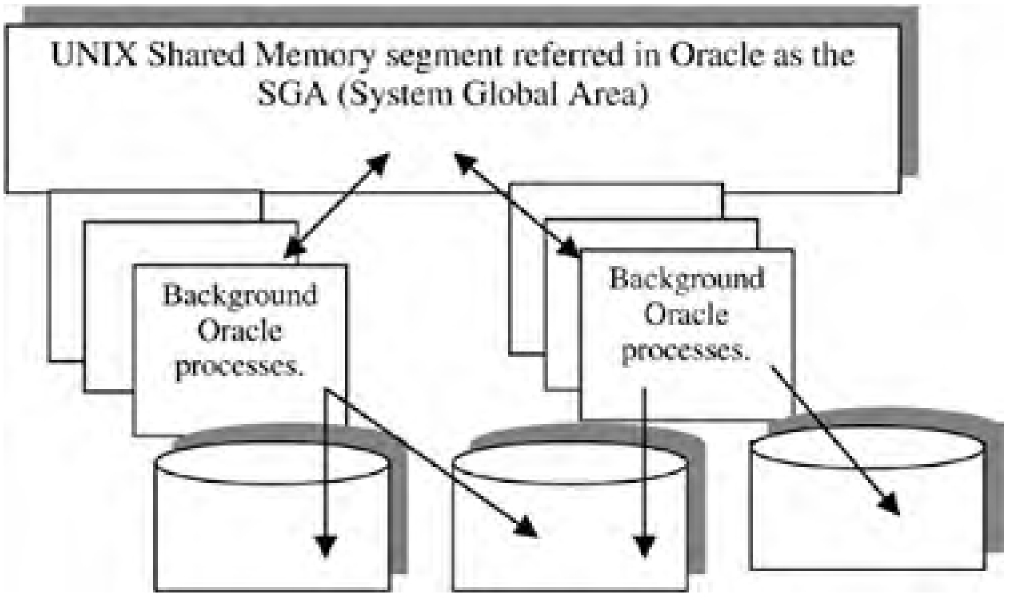
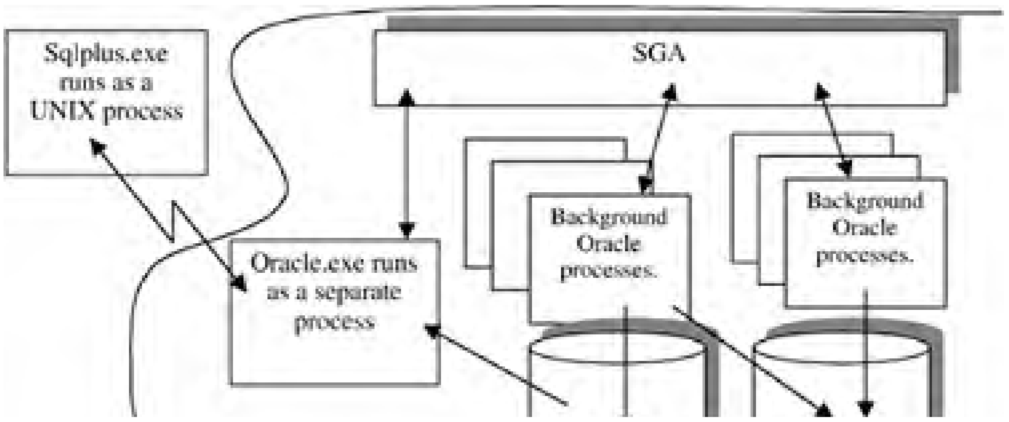

UNIX Shared Memory segment referred in Oracle as the SGA (System Global Area)
Background Oracle processes.
Background Oracle processes.

Figure 2-6. Database Instance with Connections.


Sqlplus.exe runs as a UNIX process
SGA
Oracle.exe runs as a separate process
Background Oracle processes.
Background Oracle processes.

When you invoke SQL*Plus you start an interactive program, or process, that will idle, waiting for you to type a command—SQL*Plus spends most of its time in an idle state waiting for you to type in a SQL statement. Once you establish a connection, there are actually two processes: one is the SQL*Plus program to which you are interfacing, and the other is an Oracle process to which you have a point-to-point connection. This connectivity is illustrated in the Figure 2-4 where we see SQL*Plus as a separate process, dedicated to servicing the SQL statements that you type—this other process is Oracle.

Figure 2-4 depicts three users each running SQL*Plus. Each is running a command line SQL application that has a dedicated connection to an Oracle process. For each of these point-to-point scenarios, the partnering Oracle process begins to run when the user invokes SQL*Plus with a username and password.

When SQL*Plus accepts a username, password, and host string, it attempts to connect to an Oracle database instance. Once the connection is made and the username and password are validated, the separate Oracle process is established on behalf of that SQL*Plus session.

Database instance is the term used to describe the real-time processing framework of a running database. This framework consists of Oracle background processes that communicate with each other through shared memory.

Prior to any users connected into Oracle, we have a database instance that is open, and which graphically (see Figure 2-5) means we have three main components:

1. Numerous background processes.
2. A large chunk of shared memory called the System Global Area (SGA), which can range upward from 25M.
3. Disk space, which consists of many types of files. We refer to the complete set of files on disk as "the database."

Figure 2-5 illustrates a database instance that is an open database to which users can connect. Figure 2-5 just shows the database instance and database files—no users are connected in this illustration.

Keep in mind that Figure 2-5 is a generalization of the Oracle architecture and it is generalized so as to not cloud this issue of connectivity with the complex framework of an Oracle instance. The Oracle architecture is a topic usually reserved for books on Oracle database administration.

Assume we have an Oracle instance open and no users are connected. First, we start SQL*Plus as an interactive program that initially runs as a stand-alone program. It is stand-alone until you give it an Oracle username, password and host string—then there is some activity. SQL*Plus then sends a connection request to a specific Oracle process, the Oracle listener. The Oracle listener is a host process, or Windows service, dedicated to servicing inbound connection requests. If the username and password are valid, the listener sends back a successful return code. A dedicated Oracle process is invoked, on behalf of the end user, and we have the environment illustrated in Figure 2-6. The listener is no longer involved—it only services the establishment of new incoming connections.

The "Background Oracle processes" and the SGA, in Figure 2-5, make up the processing framework of an open database—this we call the instance. The datafiles, which include a variety of Oracle files serving different purposes, are what we refer to as "the database." Your manager could say, "Make a copy of this database and send it to California." You would make copies of all the datafiles, send them to California, and copy the files onto a server where Oracle was installed. Once that was done, you could "bring up" this new instance.

The discussion surrounding Figures 2-4, 2-5, and 2-6 is designed to crystallize what happens during a database connection. Even though this is a chapter on SQL*Plus, the mechanics of connecting to Oracle are the same for any application—the underlying connection process of a .Net program to Oracle, or a JSP application using JDBC to Oracle, is identical.

There are variations to the architecture just described. For example, Oracle has an operating environment called Multi-Threaded Server (MTS) where users share Oracle processes. If you look at Figure 2-4 you might wonder about the load on a server once you approach hundreds of concurrent connections. The MTS environment is an architecture optimized for systems servicing many concurrent end users.

Once you have launched SQL*Plus and are connected to a database instance, you can submit SQL statements for processing. You can disconnect, in which case your point-to-point database connection no longer exists, and from this point you can then reconnect to the same database, connect to another database, or exit the SQL*Plus desktop application altogether.

2.4 Disconnecting

There is a difference between "disconnecting from the database" and exiting the SQL*Plus application. When you use the EXIT command:

1. Any database changes you submitted, such as INSERT statements, are committed; that is, they are permanently applied to the database. This is because SQL*Plus does an implicit commit when you end your session.
2. Your point-to-point session is terminated.
3. Finally, your SQL*Plus program exits and you are back to your desktop or command shell.

From within SQL*Plus, you do have the option to just DISCONNECT from your current database connection but keep the SQL*Plus application up. When you disconnect:

1. Any database changes you submitted, such as INSERT statements, are committed; that is, they are permanently applied to the database.
2. Your point-to-point session is terminated.
3. You are still within the SQL*Plus application but you no longer have a connection to a database. From this point you can reconnect to the same database, connect to a different database, or exit from SQL*Plus with the exit command.

Whether you DISCONNECT or you EXIT, SQL*Plus does an implicit commit.

2.5 Command Line

When you are typing a SQL statement and need additional text lines, hit Enter/Return. SQL*Plus will look at the very last (nonblank) character. If that character is a semicolon, the SQL statement is processed; otherwise, SQL*Plus assumes you are still typing the text for a single SQL statement and need another line. There is no dash or backslash needed to continue typing a SQL statement onto a next line.

A SQL statement ends with a semicolon. When you end a line with a semicolon, SQL*Plus sends the SQL statement to the database server for processing. The results are sent back and displayed in your SQL*Plus session. If it is an UPDATE statement or the result set of the SELECT statement, these results may include some information as to how many rows were updated. In the following example we see that SQL*Plus acknowledges that one row is updated. This very same information is available within embedded SQL when we use: JDBC, ODBC, Net8, or PL/SQL. In PL/SQL we can reference, following an UPDATE, the implicit cursor attribute SQL%ROWCOUNT to determine which rows were affected. SQL*Plus provides automatic feedback showing the number of rows updated. In the following code, SQL*Plus acknowledges that one row is updated.

```
SQL> UPDATE students SET college_major='Biology'
  2  WHERE student_name='John';

1 row updated.
```

2.6 Changes on the Command Line

The CHANGE command is very simple and often simpler and less time-consuming than using a mouse cut-and-paste. The actual command is CHANGE but can be abbreviated with the character "C."

Any text editor change command requires a field terminator and, in this case, the terminating character is a forward slash—by default it is a forward slash but any character can be used, as we will illustrate. The CHANGE command syntax is:

```
C[HANGE]/old_text/new_text[/]
```

Use the following to just remove text:

```
C[HANGE]/old_text
```

In this and subsequent sections, a STUDENTS table is used to demonstrate SQL*Plus. A STUDENTS table can be created using the SQL provided in Chapter 1, Section 1.6, "Behind Tables."

A helpful SQL*Plus command is the DESCRIBE command (DESC), which describes a table. The DESCRIBE command, first of all, is a quick way to see if a table exists. You get a harmless error if you attempt to describe a nonexistent table. If the table does exist, the command displays each column name, column type, plus a "Null" indicator to tell you if the column is a mandatory column—mandatory column constraints are discussed in Chapter 3. The following session text starts with a DESCRIBE command and follows with some INSERT and SQL*Plus CHANGE commands. Editorial comments are in italics and underlined. Each underlined comment precedes what that comment is describing.

```
SQL> desc students

 Name                  Null?    Type
 --------------------- -------- ------------
 STUDENT_ID                     VARCHAR2(10)
 STUDENT_NAME                   VARCHAR2(30)
 COLLEGE_MAJOR                  VARCHAR2(20)
 STATUS                         VARCHAR2(20)
```

Select table row count.

```
SQL> SELECT COUNT (*) FROM students;

  COUNT(*)
----------
         0
```

Insert a row.

```
SQL> INSERT INTO students VALUES
  2  ('A101', 'John', 'Biology', 'Degree');

1 row created.
```

We just inserted one row for the student John. We can type another INSERT statement, but for this exercise, we'll CHANGE the SQL statement in the SQL*Plus buffer. When we complete our CHANGE, we will LIST the contents of the SQL*Plus buffer and execute (using the forward slash) the SQL statement.

Change STUDENT_ID.

```
SQL> c/101/102/
  2* ('A102', 'John', 'Biology', 'Degree')
```

Change "John" to "Mary".

```
SQL> c/John/Mary
  2* ('A102', 'Mary', 'Biology', 'Degree')
```

For field separator use "." Change major.

```
SQL> c.Biology.Math/Science.
  2* ('A102', 'Mary', 'Math/Science', 'Degree')
```

*LIST SQL*Plus buffer (lower case L).*

```
SQL> l
  1  insert into students values
 2* ('A102', 'Mary', 'Math/Science', 'Degree')
```

Execute statement, then select all rows.

```
SQL> /
1 row created.

SQL> SELECT * FROM students;

STUDENT_ID STUDENT_NA COLLEGE_MAJOR       STATUS
---------- ---------- -------------------- ------
A101       John       Biology              Degree
A102       Mary       Math/Science         Degree

SQL>
```

This SQL*Plus session introduces a few topics in addition to the basic CHANGE command.

- SQL*Plus maintains a local edit buffer—big enough only for a single SQL statement. SQL*Plus keeps the last SQL statement you typed in its local edit memory buffer.
- If you mistype a SQL statement, that mistyped statement is in your local buffer and you can change what you typed because you are editing the contents of that buffer.
- You can always execute the current SQL statement by typing the forward slash.
- You can always LIST the contents of the edit buffer with the LIST (abbreviated with the letter "L") command.

A common typing error is to transpose the "R" and "F" (they are adjacent on the keyboard) when typing the "FROM" keyword of a SELECT statement. For example,

```
SQL> SELECT * FORM students;
SELECT * FORM students
         *
ERROR at line 1:
ORA-00923: FROM keyword not found where expected
```

Make correction. We do not need a forward slash at the end.

```
SQL> c/FORM/FROM
  1* SELECT * FROM students

SQL> /

STUDENT_ID STUDENT_NA COLLEGE_MAJOR       STATUS
---------- ---------- -------------------- ------
A101       John       Biology              Degree
A102       Mary       Math/Science         Degree
```

The following is a summary of common SQL*Plus editing commands discussed so far.

- The change command is CHANGE but you can just use the single character "C."
- Forward slash is the default field separator but you can use any character—the aforementioned examples used a period because a forward slash is a character within the text being changed.
- You can leave off the ending forward slash following the new-text field—shown in the immediately preceding example.
- You can always change what you just typed or what you just executed—that SQL statement is in the SQL*Plus buffer for you to edit or resubmit.
- You can LIST the current SQL statement with the LIST (just use the letter "L") command.

- You can always run whatever you are editing with the forward slash. This has nothing to do with the change command. The forward slash is a command unto itself that means: Execute the last SQL statement or whatever is in the SQL*Plus buffer.

Two forthcoming points on the CHANGE command:

- You can use CHANGE to remove text within a SQL*Plus statement.
- When typing a multiline SQL statement, you can CHANGE any line but you must first set your "current line" to the line being changed.

You remove text by not typing a NEW_TEXT clause. First, the CHANGE command definition with this consideration (notice NEW_TEXT is wholly in brackets) makes it optional.

```
C[HANGE]/old_text[/new_text[/]]
```

Because the entire NEW_TEXT clause is optional, the OLD_TEXT is removed when there is no NEW_TEXT. For example:

```
SQL> SELECT student_id, student_name FROM students;

STUDENT_ID STUDENT_NAME
---------- ------------
A101       John
A102       Mary
```

Use change command to drop "student_id," then execute.

```
SQL> c/student_id,
  1* select  student_name from students
SQL> /

STUDENT_NAME
------------
John
Mary
```

CHANGE can be used to remove additional characters when we mistype. The following removes an extra comma.

We have an extra comma after "student_name."

```
SQL> SELECT student_name, FROM students;
SELECT student_name, FROM students
                     *
ERROR at line 1:
ORA-00936: missing expression
```

Make the correction and resubmit the query.

```
SQL> c/,
  1* select student_name from students
SQL> /

STUDENT_NAME
--------------------
John
Mary
```

Multiline editing is accomplished by first identifying the line number you intend to edit. The CREATE TABLE command, in the beginning of Section 1.8, is a CREATE TABLE statement that was typed using five lines. Entering carriage returns as we type is done to make it easier to read the statement we are typing. Lengthy, multiline SQL statements can easily be saved in a SQL script file for later use. If you're comfortable with SQL and you type well, you may often find the need to type a lengthy SQL statement to quickly get to some information that is of immediate need; in other words, you're in a big hurry. The following example is a scenario of correcting a multiline SQL statement against a data dictionary view.

"column_type" should be "data_type," we expect an error.

```
SQL> SELECT column_name, column_type
  2  FROM user_tab_columns
  3  WHERE table_name='STUDENTS';
SELECT column_name, column_type
                    *
ERROR at line 1:
ORA-00904: invalid column name
```

Chapter 1, Section 1.8, introduced the Oracle data dictionary that stores information about the tables we create. It also stores a great deal of information besides tables, such as privileges that users have. We have created a table called STUDENTS. We can query the data dictionary view USER_TAB_COLUMNS, as in Section 1.8, to see what columns are defined in the data dictionary.

Typing a multiline SQL statement in SQL*Plus always displays a next-line number prompt. Earlier, SQL*Plus displayed the "2" before "FROM," and displayed "3" before "WHERE."

The aforementioned SQL statement needs a correction. We made a mistake with COLUMN_TYPE. We should have checked the definition of the view USER_TAB_COLUMNS with a DESCRIBE command. If we do this now we see that the column is not COLUMN_TYPE but DATA_TYPE.

```
SQL> desc user_tab_columns
 Name                          Null?    Type
 -------------- -------------- -------- -------------
 TABLE_NAME                    NOT NULL VARCHAR2(30)
 COLUMN_NAME                   NOT NULL VARCHAR2(30)
 DATA_TYPE                              VARCHAR2(106)
 etc . . . . . . . .
 USER_STATS                             VARCHAR2(3)
 AVG_COL_LEN                            NUMBER
```

The SQL statement is still in the SQL*Plus buffer. Go to the first line—that is where the error is—and CHANGE the text. At the SQL*Plus prompt, we type the character for the numeral "1." SQL*Plus responds with the text from the first line.

The text typed is the numeral one, not letter "L."

```
SQL> 1
  1* SELECT column_name, column_type
```

Change command to make correction.

```
SQL> c/column_type/data_type
  1* select column_name, data_type
```

*List SQL*Plus buffer, then execute the SQL statement.*

```
SQL> l
  1  SELECT column_name, data_type
  2  FROM user_tab_columns
  3* WHERE table_name='STUDENTS'
SQL> /

COLUMN_NAME                    DATA_TYPE
------------------------------ ---------
STUDENT_ID                     VARCHAR2
STUDENT_NAME                   VARCHAR2
COLLEGE_MAJOR                  VARCHAR2
STATUS                         VARCHAR2
```

What about inserting additional lines of text? For example, we see earlier, from the DESCRIBE on USER_TAB_COLUMNS, there is a column called AVG_COL_LEN. This looks interesting. Modify the query to include this column as well. Use the SQL*Plus INSERT (just use the first character "I") command. We want to insert our additional text after line 1. We can accomplish this by typing a one ("1"), then "I" followed by the new text—the change is complete.

Type LIST (L) command.

```
SQL> l
  1  SELECT column_name, data_type
  2  FROM user_tab_columns
  3* WHERE table_name='STUDENTS'
```

Type numeral one.

```
SQL> 1
  1* SELECT column_name, data_type
```

INSERT ("I"), new text.

```
SQL> i ,avg_col_len
```

Type LIST (L) command then execute with "/".

```
SQL> l
  1  SELECT column_name, data_type
  2  ,avg_col_len
  3  FROM user_tab_columns
  4* WHERE table_name='STUDENTS'
SQL> /

COLUMN_NAME                    DATA_TYPE  AVG_COL_LEN
------------------------------ ---------- -----------
STUDENT_ID                 VARCHAR2          4
STUDENT_NAME                   VARCHAR2          4
COLLEGE_MAJOR                  VARCHAR2         10
STATUS                     VARCHAR2          6

SQL>
```

Does your query show no numbers under the AVG_COL_LEN? This column shows the average column length and is a vital statistic that is used by the Oracle Cost Based Optimizer. If you have no statistics, you can generate the statistics for your STUDENTS table with the following command and rerun your query—this time you'll see the statistics. One method for gathering statistics on the STUDENTS table is with the analyze command (the PL/SQL package, DBMS_STATS, is a more robust form of gathering statistics but ANALYZE, in this case, is a straightforward solution.)

```
SQL> ANALYZE TABLE students COMPUTE STATISTICS;

Table analyzed.

SQL>
```

Now run the aforementioned query and see values in the AVG_COL_LEN column.

We added one line of text and inserted the text between lines 1 and 2. Suppose you want to add several lines. For this just the INSERT ("I") command and then Enter/Return—there is no additional text. You are then in a form of "input mode." You can type as many additional lines as you like. Typing Enter/Return twice takes you out of "input mode." In general, when you have multiple lines to change or add, you save time when you save the SQL statement to a script file, edit that file, and run it from SQL*Plus.

Following are two final topics about editing in SQL*Plus.

- Deleting a line— not to often used, but we'll cover it here.
- Appending text to a line— a very useful SQL*Plus command.

DEL is the command to delete a line. The previous SQL*Plus scenario inserted, after line 1, the text: ", AVG_COL_LEN." This became Line 2. Suppose we now want to delete this line. Set the current line at 2, then type DEL.

Type LIST (L) command.

```
SQL> l
  1  SELECT column_name, data_type
  2  ,avg_col_len
  3  FROM user_tab_columns
  4* WHERE table_name='STUDENTS'
```

Go to line 2.

```
SQL> 2
  2* ,avg_col_len
```

Delete this line.

```
SQL> del
```

Type LIST (L) showing deletion.

```
SQL> l
  1  SELECT column_name, data_type
  2  FROM user_tab_columns
  3* WHERE table_name='STUDENTS'
SQL>
```

When you set a "current line" by typing a numeral followed by enter/return, the text of that line is refreshed by SQL*Plus. You can then edit that line with a CHANGE or APPEND command. When you LIST the SQL*Plus buffer, the "current line" is always reset to the last line. So a LIST followed by a CHANGE command will always attempt to apply your change to the last line of the SQL*Plus buffer.

The APPEND command is very useful. Sometimes you type a SELECT statement; then you want to qualify it by appending a WHERE clause. The following query, as first written, selects column names from the USER_TAB_COLUMNS.

```
SQL> SELECT table_name, column_name
  2  FROM user_tab_columns;

TABLE_NAME                COLUMN_NAME
------------------------- -----------
STUDENTS                  STUDENT_ID
STUDENTS                  STUDENT_NAME
STUDENTS                  COLLEGE_MAJOR
STUDENTS                  STATUS
```

Let's modify this to see just those columns that are of type VARCHAR2. We simply use the APPEND command (abbreviate with "AP") to append a WHERE clause.

We are appending text to the clause: "FROM USER_TAB_COLUMNS." This requires a space plus an appended WHERE clause. The append command, which is "AP," is separated from the appended text with a space. You need two spaces—the second space is the first character of the text you are appending. The append command is:

Two spaces after "ap", then submit SQL query.

```
SQL> ap  where data_type = 'VARCHAR2'
2* FROM user_tab_columns where data_type='VARCHAR2'
SQL> /

TABLE_NAME                     COLUMN_NAME
------------------------------ ------------
STUDENTS                       STUDENT_ID
STUDENTS                       STUDENT_NAME
STUDENTS                       COLLEGE_MAJOR
STUDENTS                       STATUS
```

- You can append to any line. Just type the line number as shown with the CHANGE command and then do an APPEND.
- Ignore the semicolon when you append text to the last line. When you type a SQL statement you always end with a semicolon. Then, when you LIST the SQL*Plus buffer, you see there is no semicolon. When editing with either the CHANGE or APPEND commands, think about the fact that you are editing the contents of the SQL*Plus buffer, which does not include the semicolon.

The aforementioned example appended a WHERE clause and we had to begin with a space to detach the text with a space. You can sometimes just leave off a character when typing—in which case you simply append characters to the end of a line. The following statement, as first entered, is missing the "S" from the STUDENTS table—sometimes we just cannot remember if a table name is singular or plural.

Left off the "s".

```
SQL> SELECT * FROM student;
SELECT * FROM student
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

"ap"<space>"s"

```
SQL> ap s
  1* SELECT * FROM STUDENTS
SQL> /

STUDENT_ID STUDENT COLLEGE_MAJOR        STATUS
---------- ------- -------------------- ------
A101       John    Biology              Degree
```

```
A102      Mary    Math/Science        Degree

SQL>
```

You can enter a SQL statement into the SQL*Plus buffer without executing it, then list it, then execute it. This is not the normal sequence of events, but is sometimes used as a technique within SQL*Plus scripts where it is desirable to have the script listed, in a spool file, and then force the execution of that statement; that is, list first, then execute. You do this by not entering a semicolon and typing ENTER twice—this effectively ends the "input" mode. Notice the following has no semicolon at the end of line one.

Type statement, ENTER, LIST (L), then submit query (/).

```
SQL> SELECT * FROM students
  2
SQL> l
  1* SELECT * FROM STUDENTS
SQL> /

STUDENT_ID STUDENT COLLEGE_MAJOR       STATUS
---------- ------- -------------------- ------
A101       John    Biology             Degree
A102       Mary    Math/Science        Degree
```

The preceding SQL*Plus session enters a SQL statement without a final semicolon. When you type the first enter/return, SQL*Plus responds with a prompt for a second line of SQL text. SQL*Plus interprets the second enter/return as "we're done" and returns the SQL*Plus prompt. The SQL statement is entered into the SQL*Plus buffer but not executed. We can list the buffer and then execute it. This scenario can be implemented in a SQL*Plus script by having a SQL statement without a semicolon followed by a blank line followed by a LIST, then a forward slash execute command.

To summarize SQL*Plus editing, the two most helpful editing commands are:

- CHANGE ("C")
- APPEND ("AP")

We have also looked at

- LIST ("L")
- INPUT ("I")
- DELETE ("DEL")

Finally, to execute the SQL statement that is currently in your SQL*Plus buffer, you type a forward slash ("/"). One variation on the forward slash is the RUN command. The RUN command first lists the contents of the SQL*Plus buffer and then executes the statement. It is identical to the sequence: LIST followed by a forward slash.

2.7 Scripts

You can always save the contents of the SQL*Plus buffer to a file for reuse—just type the SAVE command and a filename. If the file already exists, SQL*Plus tells you. You can do a "SAVE filename REPLACE," which will replace the contents of an existing file. You also have the option to do a "SAVE filename APPEND," which adds your SQL*Plus buffer statement to the end of the filename specified.

Where does the saved file go? In UNIX the save file goes into the directory from which you launched SQL*Plus. In Windows, it goes to the Oracle "bin" directory, but, if you set up a shortcut and changed the start-in directory, as recommended in Section 2.1, the file is saved to a dedicated directory. The following illustrates how we save a select statement into a script file.

```
SQL> SELECT * FROM students;

STUDENT_ID STUDENT COLLEGE_MAJOR       STATUS
---------- ------- -------------------- ------
A101       John    Biology              Degree
A102       Mary    Math/Science         Degree

SQL> save students_select
Created file students_select
SQL>
```

We can also save a script using a full pathname.

```
SQL> save D:\sqlplus_scripts\students\students_select
Created file D:\sqlplus_scripts\students\students_select
SQL>
```

The file extension of a SQL*Plus saved file is always ".SQL."

The file text will be identical to what you type during your session—it is the exact text from the SQL*Plus buffer. The semicolon is dropped and a forward slash is added as a last and final line in the saved file. To illustrate, type a select statement, but do not type a semicolon. Then type forward slash to execute the SQL*Plus buffer.

```
SQL> SELECT * FROM students
  2  /

STUDENT_ID STUDENT COLLEGE_MAJOR       STATUS
---------- ------- -------------------- ------
A101       John    Biology              Degree
A102       Mary    Math/Science         Degree

SQL>
```

The aforementioned is identical to typing "SELECT * FROM students" and adding a semicolon. Either way, the actual text in a script file, from a SAVE command, will be two lines: the select statement minus the semicolon and a second line with a forward slash.

```
SELECT * FROM students    ".SQL" file line 1
/                         ".SQL" file line 2
```

The three options with the SQL*Plus SAVE command are:

- SAVE *filename*
- SAVE *filename* REPLACE
- SAVE *filename* APPEND

You execute a SQL script from SQL*Plus with the syntax: @filename. For example:

```
SQL> @students_select
```

```
STUDENT_ID STUDENT COLLEGE_MAJOR       STATUS
---------- ------- -------------------- ------
A101       John    Biology              Degree
A102       Mary    Math/Science         Degree
SQL>
```

During development, SQL*Plus scripts are used to drop and recreate all the tables of an application. Also for development, scripts are used to load test data. Scripts can be embedded within UNIX Korn Shell scripts to perform daily Oracle backups. They are used for ad hoc troubleshooting, (i.e., to show existing locks on database objects, to show performance of memory usage, and many other database metrics).

You can create a SQL script from SQL*Plus with a SAVE command, but scripts of length are usually created with a host editor. The contents of a SQL script can include one or more SQL statements, plus SQL*Plus commands. You can comment within a SQL*Plus script with a double dash. The following is a SQL*Plus script that contains two SQL SELECT statements plus two comment lines—the file is five lines.

```
-- Filename: SELECT_STUDENTS.SQL
-- Script to select STUDENTS data.
SELECT student_name, college_major FROM students;
SELECT student_id, student_name, college_major FROM students
WHERE status = 'Degree';
```

The aforementioned script would execute from SQL*Plus with:

```
SQL> @select_students
```

SQL*Plus scripts can contain SQL*Plus commands. This includes SQL*Plus formatting commands. A common SQL*Plus command often embedded in a SQL*Plus script is the COLUMN command. The SQL*Plus COLUMN command is used to restrict the number of characters, for a single column, used in the SQL output as well as designate a specific heading.

```
-- Filename: SELECT_STUDENTS.SQL
-- Script to select STUDENTS data.
COLUMN college_major FORMAT A12 HEADING 'College|Major'
COLUMN student_name  FORMAT A7  HEADING 'Student|Name'
COLUMN student_id    FORMAT A7  HEADING 'Student|ID'
SELECT student_name, college_major FROM students;
SELECT student_id, student_name, college_major FROM STUDENTS
WHERE status = 'Degree';

SQL> @select_students

Student Student College
ID      Name    Major
------- ------- ------------
A101    John    Biology
A102    Mary    Math/Science
```

A SQL statement always ends with a semicolon. A SQL*Plus command does not need to end with a semicolon. The next section illustrates other SQL*Plus commands: HEADING, PAGESIZE, TERM, and FEEDBACK.

2.8 Script Output

You redirect SQL*Plus output with a SPOOL command. You choose to "spool to a file" at the beginning of the script. You "end the spooling process" at the end of the script. Some scripts can be lengthy—they drop and recreate 50 tables and include many insert statements for test data. These types of scripts often spool output for verification purposes.

```
-- filename MY_STUDENTS.sql
spool my_students
DROP TABLE students;
CREATE TABLE students
(student_id    VARCHAR2(10),
 student_name  VARCHAR2(30),
 college_major VARCHAR2(15),
 status        VARCHAR2(20) ) TABLESPACE student_data;

INSERT INTO students VALUES
 ('A123','John','Math','Degree');

INSERT INTO students VALUES
 ('A124','Mary','Biology','Degree');
COMMIT;
spool off
```

The output from the spool file is named MY_STUDENTS.LST. It contains only the feedback from SQL*Plus. The text of the spool output file is brief and primarily shows that the statements completed successfully. Listing of MY_STUDENTS.LST gives the following five lines:

```
Table dropped.
Table created.
1 row created.
1 row created.
Commit complete.
```

By default, the spool file shows only the success or failure of the SQL statement submitted. You have several options if you want the spool file to include the SQL statement. One option is to include the SQL*Plus command SET ECHO ON at the beginning of your SQL script, and SET ECHO OFF at the end of the script. This option is the most practical because you can easily comment/uncomment these ECHO commands. Another option is to embed LIST SQL*Plus commands within your script.

The default extension for a SPOOL file is "LST" on Windows and UNIX, but "LIS" on OpenVMS. You can specify the extension of a spool file (e.g., you might want to actually spool text where the spooled text is a series of SQL statements, then run the output as a SQL script). This spool command would specify a SQL file extension.

```
SPOOL filename.SQL
```

The following script satisfies a common request: generate the table counts for all tables. The following is a SQL script that spools output consisting of SQL statements.

```
-- filename GEN_COUNTS.sql
spool GEN_COUNTS_OUT.SQL
SELECT 'select count (*) from '||table_name||';'
FROM    user_tables;
spool off
```

The output is a script of SQL statements. However, the output, which is the file GEN_COUNTS_OUT.SQL, contains some unwanted text (specifically, a heading and the feedback showing the number of rows selected). The resulting spool file output is:

```
'SELECTCOUNT(*)FROM'||TABLE_NAME||';'
------------------------------------------------------
select count (*) from STUDENTS;

1 row selected.
```

Eliminate the heading information and SQL*Plus feedback with SET TERM OFF and SET FEEDBACK OFF. The revised script is:

```
-- filename GEN_COUNTS.sql
set heading off
set pagesize 1000
set term off
set feedback off
spool GEN_COUNTS_OUT.SQL
SELECT 'select count (*) from '||table_name||';'
FROM    user_tables;
spool off
set heading on
set feedback on
set term on
```

The GEN_COUNTS.SQL script now generates just a SQL script. The following would be the contents of GEN_COUNTS_OUT.SQL

```
select count (*) from STUDENTS;
select count (*) from PROFESSORS;
```

The output script is GEN_COUNTS_OUT.SQL, which will perform row counts for each table. We can run GEN_COUNTS_OUT and get these numbers but the output will only show row counts. The table name is needed as well as the row count. The SQL generation script, GEN_COUNTS.SQL, undergoes a modification. The following script now includes a SQL SELECT FROM DUAL statement that will spool SET ECHO commands.

```
-- filename GEN_COUNTS.sql
set heading off
set pagesize 1000
set term off
set feedback off
spool GEN_COUNTS_OUT.SQL
SELECT 'set echo on' from dual;
SELECT 'select count (*) from '||table_name||';'
FROM   user_tables;
SELECT 'set echo off' from dual;
spool off
set heading on
set feedback on
set term on
```

The output file, GEN_COUNTS_OUT.SQL, will now provide us with row counts and the table names.

```
set echo on
select count (*) from STUDENTS;
select count (*) from PROFESSORS;
set echo off
```

For this example we used three SQL*Plus commands: HEADING, PAGESIZE FEEDBACK, and TERM.

- HEADING is a session setting that suppresses the heading of SELECT results. Your options are:
  - SET HEADING ON
  - SET HEADING OFF
- PAGESIZE set to 1000 will display your output as one stream of records with no breaks. PAGESIZE of 0 suppresses a heading and breaks. If you want a heading followed by a long report with no breaks, then set PAGESIZE to a large number and SET HEADING ON.
- FEEDBACK is also a session setting that suppresses such messages as those that indicate the number of rows that were updated or selected. Your options are:
  - SET FEEDBACK ON
  - SET FEEDBACK OFF

- TERM is a useful feature that suppresses SQL*Plus output when used in conjunction with the SPOOL command. When you set TERM OFF inside a script and you are spooling output, the results are in your spool file—only. This is a valuable feature for scripts that spool output and contain many SQL statements. Your options are:
    - SET TERM ON
    - SET TERM OFF

2.9 Command Line Arguments

You can use positional parameters with the symbols: &1, &2, &3, and so forth. Here is an example of a SQL script that takes two parameters. The intent here is for these arguments to be passed on the command line; first, the source code of the script (a total of five lines including an initial comment line), then the session that runs it.

```
-- filename MY_QUERY.sql          my_query.sql line 1
set verify off
SELECT * FROM students
WHERE student_name = '&1'
AND college_major = '&2';        my_query.sql line 5

SQL> @my_query John Business      Run with 2 parameters

STUDENT_ID STUDENT COLLEGE_MAJOR       STATUS
---------- ------- -------------------- ------
A101       John    Biology              Degree
A102       Mary    Math/Science         Degree
SQL>
```

- The command SET VERIFY OFF is a SQL*Plus session command that merely suppresses the "before" and "after" substitution values. You get "cleaner" output with verify off.
- The parameters are exact substitutions. If the parameter expression you pass needs to be in quotes within the SELECT statement, then embed the substitution parameter (&1 and &2) within single quotes.

The arguments we passed on the command line are "John" and "Business." These are in "Initial Caps." String comparisons within SQL are case sensitive; that is, "Business" is not the same as "BUSINESS." We can get around this. We can relieve the end user from being concerned with case by using SQL functions: INITCAPS(), which converts a string to initial caps, UPPER() and LOWER().

The parameters &1 and &2, embedded in single quotes, can further be embedded as arguments to an Oracle SQL function. The following is a scenario that places a command line argument within single quotes and further, within an Oracle SQL function, UPPER().

The following example is a parameter-driven data dictionary query. Data dictionary views, such as the data dictionary view USER_TAB_COLUMNS, store values in all upper case. You want to know all table and column names that contain a column named STATUS, or possibly with a name similar to STATUS. The following SQL statement is a starting point.

```
SELECT table_name, column_name FROM user_tab_columns WHERE column_name LIKE '%STATUS%';
```

This SQL query can be generalized into a parameter-driven script. To do this, first make the column name a command line argument. Secondly, alleviate the need for the user to type in upper case. Use the built-in function upper case.

```
-- filename QUERY_V1.sql           query_v1.sql line 1
set verify off
SELECT table_name, column_name
FROM   user_tab_columns
WHERE  column_name LIKE UPPER('%&1%'); query_v1.sql line 5
```

The aforementioned script, QUERY_V1.SQL, allows a user to look for any table and column with a column name like "STATUS." The user can type "Status," "status," or "STatus."

```
SQL> @query_v1 status
```

The symbol notation (&1, &2, etc.) is used for positional parameters passed on the command line. You can insert named parameters and be prompted for their value. This script performs the same query.

```
-- filename QUERY_V2.sql         query_v2.sql line 1
set verify off
SELECT table_name, column_name
FROM   user_tab_columns
WHERE  column_name
LIKE UPPER('%&column_name%');      query_v2.sql line 6
```

Execution of the aforementioned script by typing the script name only prompts you for "column_name" parameter value. Following the prompt, we enter "status."

```
SQL> @query_v2
Enter value for column_name: status

TABLE_NAME                    COLUMN_NAME
----------------------------- -----------------------------
STUDENTS                      STATUS
```

What if we have positional parameters, such as &1 and &2, and do not pass values on the command line? In that case we are prompted with a string that requests the input value for the argument names "1," "2," and so forth. The following is the result of running the first query without passing command line arguments. We are prompted for the parameter value.

```
SQL> @query_v1
Enter value for 1: status

TABLE_NAME                  COLUMN_NAME
-------------------------- ------------------------------
STUDENTS                    STATUS
```

You can provide your own prompt. The following, QUERY_V3, includes a SQL*Plus ACCEPT statement that specifically asks for the value of the "column_name" parameter.

```
-- filename QUERY_V3.sql           query_v3.sql line 1
set verify off
accept column_name prompt 'Enter a column name: '
SELECT table_name, column_name
FROM   user_tab_columns
WHERE  column_name
LIKE UPPER('%&column_name%');        query_v3.sql line 6

SQL> @query_v3
Enter a column name: status

TABLE_NAME               COLUMN_NAME
------------------------ -----------
STUDENTS                 STATUS
```

Names within a script, such as "&column_name," make the script more readable. If you use positional parameters because you often run the script with command line arguments, you can still make the script more readable by assigning the positional parameters to named parameters. This achieves the objective: command line arguments and formal parameter names like "&column_name." You can use both when you DEFINE a formal, more meaningful parameter name with the value of a positional parameter. The following script, uses "column_name" within the body, but the script immediately assigns to this variable the value of the first positional parameter.

```
-- filename QUERY_V4.sql           query_v4.sql line 1
set verify off
define column_name=&1
SELECT table_name, column_name
FROM   user_tab_columns
WHERE  column_name LIKE UPPER('%&column_name%');
```

SQL*Plus parameters can be passed as arguments to a PL/SQL procedure. The following procedure has two parameters. The first parameter is a string; the second is a number.

```
PROCEDURE my_procedure (v_param_1 VARCHAR2,
                        v_parm_2 NUMBER)
```

We also have a script that accepts two parameters: the first is a string, the second a number. From within the body of the SQL*Plus script, we can invoke our procedure, passing the values from the command line. Keep in mind that we are passing a literal expressing to our procedure, which means that if we pass STATUS (using no quotes) and 4 on the command line, this is the same as invoking the stored procedure with

```
my_procedure (STATUS, 4)
```

This is not valid. For STATUS to be a literal string, it must be in single quotes. That is, we want the following expansion:

```
my_procedure ('STATUS', 4)
```

We must code the following syntax within the SQL*Plus script:

```
'parameter_name'
```

The SQL*Plus call to the PL/SQL procedure is:

```
my_procedure ('parameter_name', etc)
```

When passing parameters to where a string is expected, always further embed the parameter within single quotes. Below is a SQL*Plus script that embeds the first command line argument within single quotes. A stored procedure is invoked from within a SQL*Plus script with the SQL*Plus EXECUTE command.

```
-- filename QUERY_V5.sql            query_v5.sql line 1
set verify off
define column_name=&1
define other_parameter=&2
. . . SQL*Plus statements
execute my_procedure('&column_name', &second_parameter)
. . . SQL*Plus statements
```

We can execute the SQL*Plus script with the following:

```
SQL> @query_5 status 120
```

The SQL*Plus script now invokes the stored procedure with the proper parameters.

```
my_procedure ('status', 120)
```

The following topics will be covered.

- SET SERVEROUTPUT ON
- DBMS_OUTPUT
- EXECUTE

SET SERVEROUTPUT ON is a SQL*Plus command that is required just once per SQL*Plus session. If you do not type this on the SQL*Plus command line, you will not see output from DBMS_OUTPUT.

From SQL*Plus you can execute any procedure, function, or package program by typing EXECUTE and the program name.

The DBMS_OUTPUT package has several procedures, but the procedure you most frequently use to print data, numbers, or text is PUT_LINE. To illustrate, the following is a SQL*Plus script that selects data from the STUDENTS table. This script returns students that have a particular major or have a particular status.

```
-- filename QUERY_V6.sql query_v6.sql line 1
set verify off
set serveroutput on
set feedback off

define p_major=&1
define p_status=&2

execute dbms_output.put_line('Parm 1='||'&p_major');
execute dbms_output.put_line('Parm 2='||'&p_status');

SELECT * from students
WHERE college_major=('&p_major')
OR status=('&p_status');
```

Passing as parameters, "Math" for the college major and "Degree" for a student status:

```
SQL> @query_v6 Math Degree
Parm 1=Math
Parm 2=Degree

STUDENT_ID STUDENT_NA COLLEGE_MAJOR        STATUS
---------- ---------- -------------------- ------
A101       John       Biology              Degree
A102       Mary       Math/Science         Degree
```

- You will notice a SQL*Plus command, SET FEEDBACK OFF. This is a SQL*Plus command that suppresses SQL*Plus messages.
- The calls to DBMS_OUTPUT.PUT_LINE() use the concatenation operator "||". This is the operator to perform string concatenation.

2.10 SQL*Plus with Korn Shell

There are three topics covered in this section.

- Invoking SQL*Plus from Korn Shell
- Invoking Korn Shell from SQL*Plus
- Passing arguments between languages

To embed SQL*Plus within Korn Shell, use Korn Shell Here-Documents. This Korn shell language feature allows "input redirection" to specify "in-stream text." The syntax uses the symbol "<<," which is followed by any character string—our example uses "EOF". Following "<< EOF" is the input-stream text consisting of SQL*Plus command file.

The following shows just a Korn Shell script that prints the first two arguments.

```
#!/bin/ksh
# KSH script to echo to parameters.
echo $1
echo $2
# end of script
```

Korn shell precedes positional parameters with a "$" while SQL*Plus uses "&". We can embed a SQL*Plus script within a Korn Shell script and run the Korn Shell script, passing arguments on the command line that are then passed to the SQL*Plus script.

The following Korn shell script accepts three arguments: username, password, and a string used to match column names. The column_name has scope within the SQL*Plus script.

```
#!/bin/ksh
# KSH script filename: script_01.ksh
username=${1}
password=${2}
column_match=${3}
sqlplus –s ${username}/${password} << EOF
SELECT table_name, column_name
FROM user_tab_columns
WHERE column_name LIKE UPPER ('${column_match}');
exit
EOF
# end of script
```

- We need to exit from SQL*Plus—to take us out of SQL*Plus and back to the Korn Shell. Thus we include EXIT, which is the last SQL*Plus statement within this script.
- The "-s" is a "Silent Mode" option that suppresses superfluous messaging by SQL*Plus.
- Within the SQL*Plus script, the Korn Shell parameter notation ($) is used.

SQL*Plus scripts can be encapsulated into distinct Korn Shell functions. The following table is a description of the Korn Shell functions used in the script.

Korn Shell Function	Description
the_tablenames_are()	This generates a list of table names.
gen_table_report()	For each table name argument, this function displays all column names.
main()	This is where execution starts. This portion of the code pipes the output from the function "the_tablenames_are" into the function "gen_table_report."

```
############################################
#
# FUNCTION: the_tablenames_are
#
#---------------------------------------------
function the_tablenames_are
{
sqlplus -s $username/$password << EOF
set feedback off
set pagesize 0
set echo off
select table_name from USER_TABLES;
exit;
EOF
}
############################################
#
# FUNCTION: gen_table_report
#
#---------------------------------------------
function gen_table_report
{
sqlplus -s $username/$password << EOF
set feedback off
set pagesize 0
set echo off
SELECT table_name, column_name FROM
USER_TAB_COLUMNS WHERE table_name = '$1';
exit;
EOF
}
############################################
#
# MAIN program code
#
#---------------------------------------------
username=${1}
password=${2}

the_table_names_are | while read tn
do
   gen_table_report $tn
done
#
############################################
```

SQL*Plus can "host out" to an operating system command using the SQL*Plus host command. Within a SQL*Plus script, you can include the SQL*Plus HOST command followed by any host command. The following is a revision of the aforementioned KSH script—this is just the SQL*Plus portion. This generates a spool file and immediately opens the spool file in an editor window.

```
SPOOL output
SELECT table_name, column_name
FROM user_tab_columns
WHERE column_name LIKE UPPER ('${column_match}');
SPOOL off
HOST vi output.lst
```

When you run the aforementioned script, the output immediately pops up for viewing. The same can be done in Windows using: HOST NOTEPAD OUTPUT.LST.

2.11 Batch Command Files

The previous section illustrates SQL*Plus scripts totally embedded within another scripting language, in this case Korn Shell. You can alternatively invoke SQL*Plus as an executable program passing as arguments: the username, password, and connect string.

The following is a host script file that will invoke SQL*Plus. Section 2.9, "Command Line Arguments," contains a SQL script, QUERY_V1.SQL, that accepts a single command line parameter. This same script can be invoked in a batch script, shown next.

```
echo "host scripting language"
sqlplus scott/tiger@ora10 @c:\my_sqlplus\query_v1.sql
status
exit
echo "host scripting language"
```

If you do not have an EXIT within the SQL*Plus script, and that certainly is one option, then you will need an EXIT following the invocation of SQL*Plus. The EXIT is needed to take you out of SQL*Plus and back into the original shell.

2.12 SQL*Plus with Active State Perl

Perl is another language that integrates well with SQL*Plus scripts. The following is a Perl program that invokes the SQL*Plus script, Query_V1.SQL.

```
#####################################################
#
# Script Filename: query_v1.pl
#
# run this program with CMD> perl query_v1.pl
#####################################################
use Win32;
$Str1       = "sqlplus scott/tiger\@ora10 ";
$Str2       = "\@c:\\my_sqlplus\\query_v1 ";
$Argument1  = "status";

$Program = $Str1.$Str2.$Argument1;
system( "start $Program" );
print "end of program \n"
```

You can embed the invocation of Perl scripts within batch scripts. You can invoke SQL*Plus from batch scripts or from Perl. You have many options to intermix scripting languages with SQL*Plus.

2.13 Privileges

The majority of scripts you develop will fall into one of two categories:

- Scripts to execute a specific function, such as loading test data into a table.
- Reporting scripts (e.g., a report on how much space your tables and indexes are using).

Report scripts tend to bring in to question your database privileges, particularly with regard to querying the data dictionary. There are three general categories of data dictionary views—these categories are designated with the prefixes: "USER," "ALL," and "DBA"—these categories are fully described in Chapter 5. In addition to these views, you also have the Oracle "real-time" performance views. The performance views contain dynamic information that is essentially "real-time" data on the current instance.

It is reasonable to write scripts that query the data dictionary and if you have the most basic Oracle roles, CONNECT and RESOURCE, you will have some limitations.

To query the data dictionary beyond the scope of the basic views—those that have the "USER" prefix—you need one of the following:

You can have the DBA role, which may be unobtainable if the database is a production or test database. If the database is your own desktop database or a development environment, you can issue the following grant:

```
GRANT DBA TO username;
```

Even when you are experimenting with your own personal database, it is unwise to connect as SYSTEM of SYS to do regular development. These two accounts certainly give you open access but, still, it is not a good idea to use these on a regular basis.

Two other options are:

```
GRANT SELECT_CATALOG_ROLE TO username;

GRANT SELECT ANY TABLE TO username;
```

Any of the aforementioned grants allows you to write unrestricted scripts against the data dictionary. The following script illustrates a useful query against the dynamic view V$INSTANCE. This script reminds you what database you are connected to—this is helpful if you connect, on a regular basis, to several databases.

```
-- Filename check_inst.sql
SET HEADING OFF
SELECT 'User='||user||'. You are connected to instance '
||instance_name ||' on '||host_name||'.'
FROM v$instance;
SET HEADING ON

SQL> @check_inst
User=SCOTT. You are connected to instance ora10 on mercury.
SQL>
```

The aforementioned script reports that we are connected to the "ora10" instance on a host machine named "mercury" and you are connected as user SCOTT.

2.14 DUAL

Literal expressions can be used within the select clause of a select statement. For example, if we have a table, PROFESSORS, with a column called SALARY, we can select all salaries plus what those salaries would be with a 10% increase.

```
SELECT SALARY, SALARY*1.1 FROM professors;
```

We can use SQL and built-in functions to do some math. To get the square root of 2, we can execute the following select statement.

```
SELECT SQRT(2) FROM professors;
```

This select will return the square root of 2 for every row in the table. If the table has just one row, then we will get the square root of 2 once. Why not create a dummy table to use with these expressions? This already exists. It is called DUAL. It is not declared in your schema, but it is in the database and every account has access to use it through a grant and a synonym. A description of the table is the following.

```
SQL> desc dual
 Name                          Null?    Type
 ----------------------------- -------- -----------
DUMMY                                   VARCHAR2(1)
```

The DUAL table is used mostly for ad hoc queries. The DUAL table with SQL*Plus provides a mechanism to experiment with built-in functions. We want to investigate the Oracle built-in function, INSTR, to see if we can pull the last forward slash from a pathname. For this exercise, hard code a test pathname. The SQL*Plus session uses "select from dual," which returns "8." This is the position of the last forward slash in the hard coded pathname.

```
SQL>
SQL> SELECT instr('aaa/bbb/ccc','/', -1,1) FROM dual;

INSTR('AAA/BBB/CCC','/',-1,1)
-----------------------------
                            8
```

Next, use this function with a SUBSTR function to extract just the file name.

```
SQL> SELECT
  2  substr('aaa/bbb/ccc',instr('aaa/bbb/ccc','/',-1,1)+1)
  3  FROM dual;

SUB
---
ccc
```

Now we can write our application code. Assume we have a PL/SQL variable with the name FULL_PATHNAME. We can assign just the filename portion to a variable with the following:

```
variable := substr(full_pathname,
               instr(full_pathname, '/', -1, 1) + 1);
```

2.15 Autotrace

AUTOTRACE is a SQL*Plus command that shows the execution plan for SQL statements. It is an excellent tool. We will see many options in Chapter 3 with regard to constraints, particularly primary key and unique constraints. These constraints have indexes. There are implications to actions that modify these constraints. In a development or test environment, developers may temporarily disable constraints or drop and create them. AUTOTRACE offers a quick check to see how your SQL will execute. You may assume that an index exists and that you are using it; and yet it may not. You learn when you run your application with disappointing performance.

Setting AUTOTRACE in your SQL*Plus session shows the execution path for all SQL statements in that session. The following demonstrates a select on the STUDENTS table, defined in the DDL of Chapter 4.

The following session sets AUTOTRACE. A SELECT statement is entered and executed. The AUTOTRACE shows that the SELECT statement is using the index.

```
SQL> SET AUTOT ON EXP
SQL> SELECT status FROM students WHERE student_id='A101';

STATUS
--------------------
Degree

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE
         (Cost=1 Card=1 Bytes=10)
   1   0   TABLE ACCESS (BY INDEX ROWID) OF 'STUDENTS'
           (Cost=1 Card=1 Bytes=10)
   2   1     INDEX (UNIQUE SCAN) OF 'PK_STUDENTS' (UNIQUE)
```

Suppose a developer disabled the constraint with the following.

```
SQL> ALTER TABLE students
  2  DISABLE CONSTRAINT pk_students CASCADE;
```

Following the aforementioned DISABLE command, our SQL trace shows a much different result. Below is the SQL Trace result—we see that now our query is performing a full table scan. From this we will want to investigate our SQL—maybe it is not written properly or maybe a constraint or index has been dropped. Had we not used this short analysis tool, we could likely put our select statement into production having it do table scans.

```
SQL> SELECT status FROM students WHERE student_id='A010';

Execution Plan
----------------------------------------------------------
   0     SELECT STATEMENT Optimizer=CHOOSE
        (Cost=1 Card=1 Bytes=10)
   1   0   TABLE ACCESS (FULL) OF 'STUDENTS'
           (Cost=1 Card=1 Bytes=10)
```

You have three types of output to consider when using AUTOTRACE.

- Do you want to execute the SQL statement? You may not if the SQL takes a long time to run. Long-running SQL statements are a reason to test with AUTOTRACE. You do have the option to see an explain plan and/or statistics without fully executing the query.
- You can get the explain plan—the most useful piece of information.
- You can also get statistics that indicates the number of physical and logical reads performed.

The AUTOTRACE syntax is:

```
SET AUTOT[RACE] {ON|OFF|TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]]
```

SQL*Plus Command	Explanation

SET AUTOT OFF	Disables AUTOTRACE for the session.
SET AUTOT ON	This command presumes EXP and STAT. • Executes the statement • Generates the explain plan • Generates the statistics
SET AUTOT TRACE	Do NOT execute the SQL statement. This also presumes EXP and STAT. • Generates the explain plan • Generates the statistics
SET AUTOT TRACE EXP STAT	Same as SET AUTOT TRACE.
SET AUTOT TRACE EXP	Do NOT execute SQL statement. • Generates the explain plan
SET AUTOT TRACE STAT	Do NOT execute SQL statement. • Generates the statistics

The AUTOTRACE option, shown here, explains the SQL statement. It does not execute the statement. This is useful for long running queries.

```
SET AUTOT ON EXP
```

You do need a plan table for this to run. Oracle populates the plan table as it analyzes the SQL statement. The script for the plan table is found in the directory:

```
ORACLE_HOME/rdbms/admin/utlxplan.sql
```

The plan table, if you need to key in, is:

```
CREATE TABLE PLAN_TABLE (
statement_id     VARCHAR2(30),
timestamp        DATE,
remarks          VARCHAR2(80),
operation        VARCHAR2(30),
options          VARCHAR2(255),
object_node      VARCHAR2(128),
object_owner     VARCHAR2(30),
object_name      VARCHAR2(30),
object_instance  NUMERIC,
object_type      VARCHAR2(30),
optimizer        VARCHAR2(255),
search_columns   NUMBER,
id               NUMERIC,
parent_id        NUMERIC,
position         NUMERIC,
cost             NUMERIC,
cardinality      NUMERIC,
bytes            NUMERIC,
other_tag        VARCHAR2(255),
partition_start  VARCHAR2(255),
partition_stop   VARCHAR2(255),
partition_id     NUMERIC,
other            LONG,
distribution     VARCHAR2(30),
cpu_cost         NUMERIC,
io_cost          NUMERIC,
temp_space       NUMERIC,
access_predicates   VARCHAR2(4000),
filter_predicates   VARCHAR2(4000));
```

Chapter Three. Declarative Constraints

This is the first of several chapters that cover enforcing business rules with constraints. Declarative constraints provide a core and traditional strategy for business rule enforcement. This chapter explains the how and why of declarative constraints. Chapter 4 illustrates the use of declarative constraints and includes a sample student's data model. Chapter 5 demonstrates how to extract declarative constraint definitions from the database data dictionary. Chapter 6 begins a series of chapters on triggers that enforce constraints procedurally.

This chapter makes frequent reference to a STUDENTS table. The STUDENTS table is part of the sample data model described in Chapter 4. The model description includes an entity relationship diagram, DDL, sample data, plus a description of the business rules enforced.

Oracle supports the following SQL constraints:

- Primary Key constraint
- Unique constraint
- Foreign Key constraint
- Check constraint
- Not Null constraint, which is really a special case of a Check constraint

3.1 Primary Key

The PRIMARY KEY of a table constrains a single column, or set of columns, to a unique and mandatory value—mandatory, meaning that no column in the primary key can ever be null. A table need not have a primary key, but this is the exception; most tables are created with a primary key.

Consider a table that stores information about students. We must be able to identify a student and store data associated with that individual student. Each student must have a row and only one row in the STUDENTS table. Also, each row in the STUDENTS table should identify one, and only one, student. The primary key mechanism enables an application, for example, to properly process student tuition bills. Every student will get one, and only one, tuition bill.

In addition to business rule enforcement, there are other database motivations. Primary keys are an integral part of parent-child relationships that enforce referential integrity. Additionally, a primary key requires an index, a physical structure consuming disk space that, when used properly, provides fast access to the data.

The DDL for the primary key constraint can be embedded within the CREATE TABLE statement. Embedded DDL has two syntax versions: a column constraint clause and a table constraint clause. The primary key constraint can be created separate from the table creation statement. This is accomplished with an ALTER TABLE statement. Creation of the constraint with ALTER TABLE can be done immediately after the table is created. It can also be done after the table is populated.

The Oracle database engine enforces the rule of the primary key constraint. Once you create a table with a primary key, you are saying that: first, all values for the primary key are unique; and second, they must have a value, which means that no column in the primary key can ever be NULL.

Enforcement of a constraint can be temporarily disabled and later enabled. This is accomplished with an ALTER TABLE statement and the constraint options: DISABLE and ENABLE. One reason to load data with a disabled constraint is to reduce the load time. The load time with a disabled constraint will be less because indexes are disabled and therefore not updated.

Within a transaction, an application can temporarily suspend constraint enforcement. In this case a program begins a transaction by setting the constraint to a deferred state. The data is loaded followed by a commit. Upon commit, the constraint is applied by Oracle. This option requires that the constraint be created with the DEFERRABLE keyword.

You can load data into a table with a constraint disabled and, after the load, enable the constraint so that the rule is applied only to new data. The old data, should it violate the business rule, can remain in the table. This strategy to business rule enforcement can apply to data warehouses that must have historical data available for analysis. This option requires that the constraint be enabled with a NOVALIDATE keyword.

Several types of primary key enforcement, such as DEFERRABLE and NOVALIDATE, will affect the type of index used with the primary key constraint. These options will use a nonunique index. A conventional primary key constraint, without ever attempting to violate it, even temporarily within a transaction, will use a unique index.

The fact that a table has a primary key is stored in the data dictionary. We have looked at the data dictionary view, USER_TAB_COLUMNS (see Chapter 1, Section 1.6.2, "Data Dictionary"), which is the dictionary view for looking at column names within a table. We also have views for constraints, these are USER_CONSTRAINTS and USER_CONS_COLUMNS. These views show what tables have constraints, as well as constraint name, type, and status.

Two methods by which you can challenge a primary key constraint is to INSERT a duplicate row or UPDATE a column value forcing a duplicate—in either case, the INSERT or UPDATE statement will fail with an Oracle error.

Declaring constraints is only part of an application. Application code should be integrated with constraint enforcement and handle errors that propagate from constraint violations. If your application is an OLTP system and data entry can cause a duplicate insert, the code should contain graceful error handling. It should produce a meaningful end-user response more meaningful than the generic text of a constraint violation or an alarming stack trace.

3.1.1 Creating the Constraint

This section covers creating the constraint with the column constraint clause, table constraint clause, and the ALTER TABLE statement. This section will use a sample table, TEMP, that has the following description:

```
Name                           Null?    Type
------------------------------ -------- ------------
ID                                      VARCHAR2(5)
NO                                      NUMBER
```

There are several syntax methods and styles for creating a primary key:

1. Column Constraint Clause
2. Table Constraint Clause

3. ALTER TABLE statement

The following discussion addresses the three styles with respect to creating a PRIMARY KEY constraint. Other types of constraints, UNIQUE, FOREIGN KEY, and CHECK, can also be created with each style.

3.1.1.1 COLUMN CONSTRAINT CLAUSE

The following creates a table, TEMP, with two columns. The column ID is the primary key. This is an example of a column constraint clause.

```
SQL> CREATE TABLE temp(id VARCHAR2(5) PRIMARY KEY, no NUMBER);

Table created.
```

Once the table is created rows are inserted; however, an INSERT with a duplicate value for the column ID will fail. The third INSERT statement is a duplicate and consequently fails.

```
SQL> insert into temp values ('AAA', 1);      First row.

1 row created.

SQL> insert into temp values ('BBB', 2);      Second row.

1 row created.

SQL> insert into temp values ('AAA', 3);      Duplicate.
insert into temp values ('AAA')
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C006083) violated
```

For the last insert, SQL*Plus flushed an error message to the display. This is the behavior of SQL*Plus. If we are writing code in Java or PL/SQL and anticipate a duplicate insert, we can write error handling code, capture the error condition, and handle it gracefully.

The duplicate insert error message prefix is "ORA". The error number is "-00001." That is not a dash between "ORA" and the number, but "ORA" and a minus one.

The aforementioned error message references "SCOTT.SYS_C006083"; this is the name of the primary key constraint. This name was internally generated by Oracle during execution of the CREATE TABLE TEMP statement. To name the primary key constraint with a column constraint clause:

```
CREATE TABLE temp
(id VARCHAR2(5) CONSTRAINT PRIMARY KEY my_constraint_name,
 no NUMBER);
```

Column constraint clauses are quick and appropriate for ad hoc SQL. Two restrictions on column constraint clauses for primary keys are: (a) no concatenated primary key can be declared and (b) you cannot stipulate the tablespace of the index created on behalf of the constraint. Both concatenated keys and tablespace clauses for indexes are covered later in this chapter.

3.1.1.2 TABLE CONSTRAINT CLAUSE

The table constraint clause is attached at the end of the table definition. Table constraint clauses are part of the CREATE TABLE statement—they just come after all the columns are defined. If there is a syntax error in the constraint clause, the statement fails and no table is created.

The following illustrates, in template form, a CREATE TABLE statement that declares a primary key. The table constraint clause allows multiple constraints to be included with a comma separating each constraint definition.

```
CREATE TABLE temp
(id  VARCHAR2(5),
 no  NUMBER,
CONSTRAINT PRIMARY KEY (id),
CONSTRAINT. . next constraint,
CONSTRAINT. . next constraint) TABLESPACE etc;
```

The following creates the TEMP table, using a table constraint clause.

```
CREATE TABLE temp
(id  VARCHAR2(5),
 no  NUMBER,
CONSTRAINT PRIMARY KEY (id)) TABLESPACE student_data;
```

3.1.1.3 ALTER TABLE STATEMENT

The ALTER TABLE statement is another option for managing constraints. Once you create a table, you can use the ALTER TABLE statement to manage constraints, add columns, and change storage parameters. The ALTER TABLE command regarding constraints is used to perform the following:

Function to Perform	ALTER Syntax
Add a constraint	ALTER TABLE *table_name* ADD CONSTRAINT *etc*
Drop a constraint	ALTER TABLE *table_name* DROP CONSTRAINT *etc*
Disable a constraint	ALTER TABLE *table_name* DISABLE CONSTRAINT *etc*
Enable a constraint	ALTER TABLE *table_name* ENABLE CONSTRAINT *etc*

The following DDL consists of two DDL statements: a CREATE TABLE statement and an ALTER TABLE statement for the primary key. In this example, the constraint is named PK_TEMP.

```
CREATE TABLE temp (id  VARCHAR2(5), no  NUMBER);

ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY (id);
```

The ALTER TABLE command has many options. An approach to remembering the syntax is to consider the information Oracle needs to perform this operation:

- You have to say what table you are altering; you begin with:

  ALTER TABLE *table_name*

- Then, what are you doing? Adding a constraint:

  ALTER TABLE *table_name* ADD CONSTRAINT

- It is highly recommended but not required that you include a name for the constraint. The constraint name is not embedded in quotes, but will be stored in the data dictionary in upper case. For the table TEMP, append the constraint name PK_TEMP.

  ```
  ALTER TABLE temp ADD CONSTRAINT pk_temp
  ```

- Denote the type of constraint that will be a PRIMARY KEY, UNIQUE, FOREIGN KEY, or CHECK constraint.

  ```
  ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY
  ```

- There are a few specific options that follow the reference to the constraint type. For a PRIMARY KEY and UNIQUE constraint, designate the columns of the constraint. For a CHECK constraint, designate the rule. The primary key for the TEMP table is created with the following syntax.

  ```
  ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY (ID);
  ```

- For PRIMARY KEY and UNIQUE constraints we should designate the tablespace name of the index that is generated as a by-product of the constraint—this topic is covered in detail in Section 3.1.3, "The Primary Key Index." To designate the index tablespace, use keywords USING INDEX TABLESPACE. The final syntax is:

```
ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY (ID) USING INDEX TABLESPACE student_index;
```

3.1.2 Naming the Constraint

This section will use a sample table, TEMP, that has the following description:

```
Name                           Null?    Type
------------------------------ -------- ------------
ID                                      VARCHAR2(5)
NO                                      NUMBER
```

The following example creates a table with an unnamed primary key constraint.

```
CREATE TABLE temp(id VARCHAR2(5) PRIMARY KEY, no NUMBER);
```

All constraints have a name and when no name is supplied, Oracle internally creates one with a syntax, SYS_C, followed by a number. The constraint name is important. Troubleshooting frequently requires that we query the data dictionary for the constraint type, table, and column name. For the aforementioned CREATE TABLE statement, a duplicate insert will produce an error.

```
ORA-00001: unique constraint (SCOTT.SYS_C006083) violated
```

The following creates the table and assigns the constraint name PK_TEMP:

```
CREATE TABLE temp
(id  VARCHAR2(5) CONSTRAINT pk_temp PRIMARY KEY,
 no   NUMBER);
```

A duplicate insert here includes the constraint name as well. This constraint name, PK_TEMP, is more revealing as to the nature of the problem.

```
ORA-00001: unique constraint (SCOTT.PK_TEMP) violated
```

Regardless of the syntax form, you can always name a constraint. The following illustrates the TEMP table and primary key constraint with a column constraint clause, table constraint clause, and ALTER TABLE statement, respectively.

```
(1) CREATE TABLE temp
    (id  VARCHAR2(5) CONSTRAINT pk_temp PRIMARY KEY,
      no  NUMBER);

(2) CREATE TABLE temp
    (id  VARCHAR2(5),
     no  NUMBER,
    CONSTRAINT pk_temp PRIMARY KEY (id));

(3) CREATE TABLE temp
    (id  VARCHAR2(5),
     no  NUMBER);
    ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY (id);
```

Having consistency to constraint names, or following a format, is just as important as naming the constraint. For all stages of a database, be it development, test, or production, it makes sense to have primary key constraints named. Primary key constraints are commonly named with one of two formats. Both formats indicate the constraint type with the PK designation and the table name.

1. PK_*table_name*
2. *table_name*_PK

Suppose you just installed a new application and your end user calls you. Would you rather have the user say, "I ran this application and got an Oracle error that says: SYS_C006083 constraint violation." Or, would you rather hear, "I ran this application and got an Oracle error: PK_TEMP."

If the constraint name is PK_TEMP, you can quickly comfort the caller because you know exactly what the problem is: a primary key constraint violation on that table. Without a meaningful constraint name, you would begin with querying the data dictionary views DBA_CONSTRAINTS to track the table upon which that constraint, SYS_C006083, is declared.

A common oversight when using a data modeling tool is to not name constraints. A data modeler is usually far more concerned with system requirements than constraint names. Having to go back and type in numerous constraint names can be tedious. If you use such a tool, it is worthwhile to do a small demo to see what DDL the tool generates. You want to make sure the final DDL meets your criteria and that you are using the features of the tool that generate the desired DDL. Oracle Designer, very conveniently, will automatically name primary key constraints with a "PK" prefix followed by the table name.

3.1.3 The Primary Key Index

This section covers the index that must always exist as part of the primary key. The topic of tablespace is included. Refer to Chapter 1, Section 1.6.1, "Application Tablespaces," for additional information on the definition, use, and creation of an Oracle tablespace.

This section will use a sample table, STUDENTS, that has the following description:

```
Name                           Null?    Type
------------------------------ -------- ------------
STUDENT_ID                              VARCHAR2(10)
STUDENT_NAME                            VARCHAR2(30)
COLLEGE_MAJOR                           VARCHAT2(15)
STATUS                                  VARCHAR2(20)
```

Whenever you create a primary key, Oracle creates an index on the column(s) that make up the primary key. If an index already exists on those columns, then Oracle will use that index.

Indexes are an integral part of the primary key. Depending on the primary key options, the index may be unique or nonunique. Deferrable primary key constraints use nonunique indexes. Indexes are not used to enforce the business rule of the primary key, but an index is still required. The benefits of the index are seen with queries against the table. If the primary key constraint is disabled, the index is dropped and query performance suffers.

Tables occupy physical storage. Indexes also use physical storage. The creation of the primary key should designate a tablespace for that index. Because of I/O contention and the fact that indexes grow differently than tables, we always place indexes in separate tablespaces.

The following ALTER TABLE statement creates the primary key, plus designates the tablespace for the index—USING INDEX TABLESPACE is a keyword phrase to this syntax.

```
CREATE TABLE students
 (student_id    VARCHAR2(10),
  student_name  VARCHAR2(30),
  college_major VARCHAR2(15),
  status        VARCHAR2(20)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;
```

If you do not designate a tablespace for the primary key index, that index is built in your default tablespace. All Oracle user accounts are created with a default tablespace. Tables and indexes created by a user with no tablespace designation fall into this default tablespace.

For example, the following DDL creates an index object in the default tablespace. Because there is no tablespace clause on the CREATE TABLE statement, the table is also created in the default tablespace.

```
CREATE TABLE temp(id VARCHAR2(5) PRIMARY KEY, no NUMBER);
```

The following puts the TEMP table in the STUDENT_DATA tablespace and the primary key in the STUDENT_INDEX tablespace.

```
CREATE TABLE temp(id VARCHAR2(5), no NUMBER)
tablespace STUDENT_DATA;

ALTER TABLE temp ADD CONSTRAINT pk_temp PRIMARY KEY (ID)
USING INDEX TABLESPACE student_index;
```

To create tables and indexes in tablespaces other than your default requires privileges. If you have the RESOURCE role, then you may still have the privilege UNLIMITED TABLESPACE—this privilege is automatically inherited with the RESOURCE role and gives you unlimited tablespace quotas. With UNLIMITED TABLESPACE you are able to create objects in any tablespace including the SYSTEM tablespace. For this reason, the privilege is often revoked from application developers and tablespace quotas are added.

The change made to developer accounts is similar to the change made to the SCOTT account as follows.

```
REVOKE UNLIMITED TABLESPACE FROM SCOTT;
ALTER USER SCOTT QUOTA UNLIMITED ON STUDENT_DATA;
ALTER USER SCOTT QUOTA UNLIMITED ON STUDENT_INDEX;
ALTER USER SCOTT DEFAULT TABLESPACE STUDENT_DATA;
```

Check your account privileges and tablespace quotas with the following SQL. Privileges and roles that are granted to your Oracle account are queried from the data dictionary views USER_ROLE_PRIVS and USER_SYS_PRIVS:

```
column role_priv format a30
SELECT  'ROLE: '||granted_role role_priv
FROM     user_role_privs
UNION
SELECT  'PRIVILEGE: '||privilege role_priv
FROM     user_sys_privs;

ROLE_PRIV
------------------------------
PRIVILEGE: SELECT ANY TABLE
PRIVILEGE: CREATE ANY MATERIALIZED VIEW
ROLE: CONNECT
ROLE: RESOURCE
ROLE: SELECT_CATALOG_ROLE
```

To see tablespace quotas, query USER_TS_QUOTAS:

```
SELECT tablespace_name, max_bytes FROM user_ts_quotas;
```

The aforementioned SQL will return a minus 1 for any tablespace for which you have an unlimited quota. For example:

```
TABLESPACE_NAME                 MAX_BYTES
------------------------------ ----------
STUDENT_DATA                           -1
STUDENT_INDEX                          -1
```

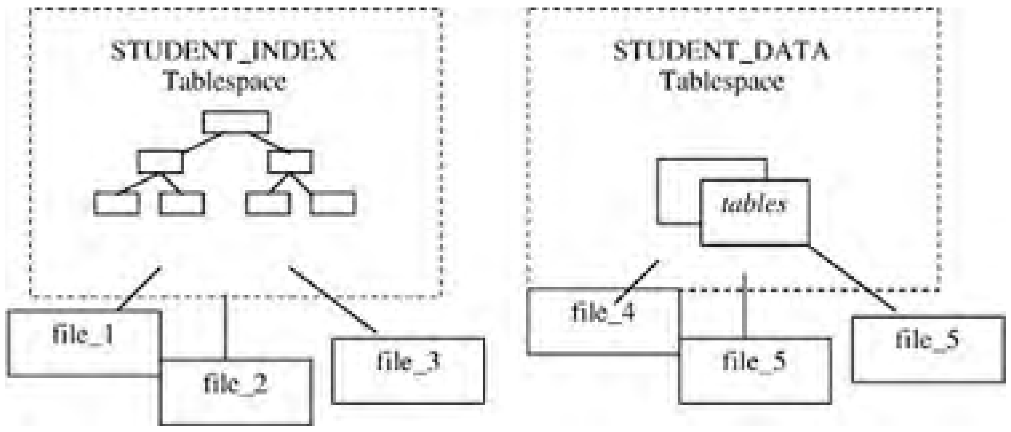
An index is an object that is created in a tablespace. It is a physical structure that consumes disk space. When you create a Primary Key or Unique constraint, an index is either automatically created or an existing index may be reused.

An index is based on a tree structure. Indexes are used by Oracle to execute SELECT statements. The execution of a SELECT using an index is generally faster than a SELECT that does not use an index.

Indexes are generally created in tablespaces separate from tablespaces for tables. Indexes from primary keys make up a portion of all indexes in an application. A tablespace for indexes will have indexes from primary key constraints, indexes from unique constraints, and indexes created to speed up selected queries.

Figure 3-1 illustrates the physical separation of the index structures and table structures. The left side of Figure 3-1 represents a tablespace for indexes. All index structures physically exist in the files for this tablespace. The right side illustrates the allocation of tables. Tables physically exist in the files for the STUDENT_DATA tablespace.

Figure 3-1. Primary Key Indexes.



To determine the physical space allocated by the STUDENTS table and the primary key index, query DBA_EXTENTS and DBA_DATA_FILES. The following illustrates a SQL*Plus session query against these views. Because these views begin with DBA, you need either the DBA role or SELECT_CATALOG_ROLE role (refer to Chapter 5, Section 5.1, "What You Can See," for additional information on data dictionary views).

```
SQ1>  SELECT a.extent_id,
  2*        a.segment_name,
  3*        b.file_name,
  4*        round(a.bytes/1024) KBytes
  5*    FROM dba_extents a,
  6*        dba_data_files b
  7*   WHERE segment_name in ('STUDENTS','PK_STUDENTS')
  8*     AND a.file_id=b.file_id;
SQL>
```

The result from the previous code will produce something like the following.

```
EXTENT_ID  SEGMENT_NAME  FILE_NAME                 KBYTES
---------  ------------  ------------------------- ------
        0  STUDENTS      D:\. .\STUDENT_DATA01.DBF      60
        0  PK_STUDENTS   D:\. .\STUDENT_INDEX01.DBF     60
```

The previous output shows that the index created from the primary key initially consumes 60K of disk space in the file STUDENT_INDEX01.DBF. This allocated space is in a file, separate from where the initial 60K of disk space is allocated for the STUDENTS table.

The SEGMENT_NAME column, PK_STUDENTS, is the name of the index, not the constraint. When you create a primary key constraint, an index is created with the same name. That is the situation discussed earlier. The index name can differ if the index is created first and then the constraint is created. When a primary key or unique constraint is created, Oracle looks for an index on the table and on the same columns of the constraint. If such an index exists, that index is used, regardless of the index name.

The following paragraphs explain the aforementioned SQL*Plus query that selects data dictionary information from two views: DBA_EXTENTS and DBA_DATA_FILES.

Objects that require disk space are also called segments. A table is a relational object, but it is also a physical segment—it uses disk space. You query DBA_SEGMENTS to get the physical attributes of tables, partitions, indexes, clusters, and materialized views—any object that requires disk capacity. Stored procedures and sequences are not segments—they are objects that are defined in the data dictionary and their definitions exist in the system tablespace. For every segment, there are one or more extents. An extent is a contiguous set of database blocks.

A table is a segment. The table segment can consume 15 contiguous blocks on disk, where each block is 4K. Blocks 1 through 15, times the 4K block size, yields 60K. This is one extent. There can be other extents. Each extent is a contiguous series of 15 blocks, each block being 4K. Block sizes can vary, and so can the size and number of extents.

The query result shows that there is 60K of contiguous blocks allocated for the index and 60K of contiguous blocks allocated for the STUDENTS table. This is what is first allocated just as a result of creating the table and index. When the 60K fills up, the table or index will extend to another 60K. The datafile for the STUDENT_DATA tablespace is 5M. We are only using 60K of that.

The data dictionary views DBA_SEGMENTS, DBA_EXTENTS, and DBA_DATA_FILES are three of many dictionary-wide views that provide detailed information about space allocation of segments.

The following summarizes the relationship between the primary key constraint and the index.

- An index can be created on any column or set of columns other than the primary key. When we execute the DDL statement that creates a primary key, regardless of the syntax, an index is always created, provided an index on those exact columns does not already exist. A primary key and unique constraint are not the only means of establishing an index. Frequently, many other indexes are created as a means to achieve optimal performance.

- The fact that there is a primary key constraint is wholly defined within the data dictionary. No space is allocated anywhere except the data dictionary tablespace that records this constraint definition. However, the index is an integral part of the constraint. It is an object, takes space, and can be viewed from the data dictionary views USER_OBJECTS and USER_INDEXES, USER_SEGMENTS, and USER_EXTENTS.

- A primary key constraint can be challenged with an INSERT and UPDATE statement—these are the only means by which we can possibly attempt to violate the constraint. The index generated by the primary key constraint does provide a valuable mechanism for optimizing SELECT statements.

3.1.4 Sequences

This section will use a sample table, STUDENTS, that has the following description:

```
Name                          Null?    Type
----------------------------- -------- ------------
STUDENT_ID                              VARCHAR2(10)
STUDENT_NAME                               VARCHAR2(30)
COLLEGE_MAJOR                              VARCHAT2(15)
STATUS                                 VARCHAR2(20)
```

The STUDENTS table with primary key constraint is created with the following DDL.

```
CREATE TABLE students
 (student_id    VARCHAR2(10),
  student_name  VARCHAR2(30),
  college_major VARCHAR2(15),
  status        VARCHAR2(20)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;
```

For our student system database, what do we use for a STUDENT_ID? One option is to generate a unique student ID for each student that enters the college. We could use social security numbers, information from student visas, even driver's license data. An alternative is for the application code to auto-generate student ID numbers as each student is added to the database.

Databases handle auto-generated columns in a variety of ways. SQL Server uses an IDENTITY column; Oracle uses sequences. We start with creating a sequence that is an object. The existence of that object is stored in the data dictionary. The sequence always has state information, such as current value, and that context remains within the data dictionary. Once you create a sequence you can query the attributes of that sequence from the data dictionary view USER_SEQUENCES.

A sequence has two attributes, NEXTVAL and CURRVAL:

Sequence Attribute	Description
sequence_name.NEXTVAL	This evaluates to the next highest value.
sequence_name.CURRVAL	This evaluates to the value that was returned from the most recent NEXTVAL call.

We can experiment with sequences right within SQL*Plus as illustrated with the following script. The SQL*Plus session text that follows creates a sequence named MY_SEQUENCE and then uses that sequence to insert values into a TEMP table, also created here. The first statement creates a sequence with defaults, which means that the first time we use that sequence, the value for NEXTVAL will be the number 1.

```
SQL> CREATE SEQUENCE my_sequence;
Sequence created.

SQL> CREATE TABLE temp(n NUMBER);
Table created.

SQL> INSERT INTO temp VALUES (my_sequence.NEXTVAL);
1 row created.

SQL> /                    Repeat the last insert.
1 row created.

SQL> /                    Repeat the last insert.
1 row created.

SQL> INSERT INTO temp VALUES (my_sequence.CURRVAL);
1 row created.

SQL> SELECT * FROM temp;       Now, what is in this table?
         N
----------
         1
         2
         3
         3
```

The keyword START WITH designates the starting point of a sequence. To recreate the sequence with a starting point of 10:

```
SQL> DROP SEQUENCE my_sequence;
Sequence dropped.

SQL> CREATE SEQUENCE my_sequence START WITH 10;
Sequence created.

SQL> SELECT my_sequence.NEXTVAL from dual;

  NEXTVAL
----------
       10
```

If you shut the database down, start it back up, and repeat the previous three INSERT statements, the sequence numbers will continue with 4, 5, and 6. They will continue sequencing because the state of the sequence number is maintained in the data dictionary. When the database is shut down, the state of that sequence number is somewhere in the system tablespace datafile.

Sequences can be created so that the numbers either cycle or stop sequencing at some maximum value. Keywords for this are CYCLE and NOCYCLE. The INCREMENT BY interval can create the sequence to increment by a multiple of any number; the default is 1. The CACHE option pre-allocates a cache of numbers in memory for improved performance. The following illustrates a sequence that will cycle through the values: 0, 5, 10, 15, and 20, then back around to 0.

```
SQL> CREATE SEQUENCE sample_sequence
  2* MINVALUE   0
  3* START WITH 0
  4* MAXVALUE  20
  5* INCREMENT BY 5
  6* NOCACHE
  7* CYCLE;

Sequence created.
```

Sequence numbers are not tied to any table. There is no dependency between any table and a sequence. Oracle has no way to know that you are using a particular sequence to populate the primary key of a specific table.

If you drop and recreate a sequence you may possibly, temporarily, invalidate a stored procedure using that sequence. This would occur if a stored procedure includes the sequence with NEXTVAL or CURRVAL in a PL/SQL or SQL statement. Once the sequence is dropped, the stored procedure becomes invalid. Once the sequence is recreated the procedure will be compiled and validated at run time.

Once you create a sequence you can use it to populate any column in any table. For this reason it is very desirable to carefully name sequences to reflect their use and restrict sequences to the purpose of populating a specific column.

If the primary key is generated by a sequence, the sequence should be named with the following format:

*table_name*_PK_SEQ

Using this syntax, a sequence dedicated to the generation of primary key values in the STUDENTS table would be named:

```
STUDENTS_PK_SEQ
```

The view, USER_SEQUENCES, shows the attributes of all sequences in a schema. If we see a sequence named STUDENTS_PK_SEQ, we are quite certain this sequence is used to populate the primary key column of the STUDENTS table. We are certain only because of the sequence name, not from any other data dictionary information.

The complete DDL to create the STUDENTS table, primary key constraint with tablespace clause, and the sequence is the following:

```
CREATE TABLE students
 (student_id    VARCHAR2(10),
  student_name  VARCHAR2(30),
  college_major VARCHAR2(15),
  status        VARCHAR2(20)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

CREATE SEQUENCE students_pk_seq;
```

3.1.5 Sequences in Code

You can use the NEXTVAL attribute in any SQL INSERT statement to add a new student. The following Java procedure uses the sequence STUDENTS_PK_SEQ to insert a new student. This code could be called when a student submits, as part of the admission process, an HTML form with the student name, subject major, and status. There is no student ID on the HTML form—the student ID is evaluated as part of the SQL INSERT statement.

```
public void insertStudent(String StudentName,
    String CollegeMajor, String Status)
{
    try
    {
       stmt = conn.prepareStatement
       ("INSERT INTO students (student_id," +
        " student_name, college_major, status)" +
        " VALUES( students_pk_seq.NEXTVAL, :b1, :b2, :b3) ");

       stmt.setString(1, StudentName);
       stmt.setString(2, CollegeMajor);
       stmt.setString(3, Status);
       stmt.execute();
    }
    catch (SQLException e)
    {
       if (e.getErrorCode() == 1)
       {
           System.out.println("We have a duplicate insert.");
       }
    }
}
```

Section 3.1.1, "Syntax Options," illustrates a SQL*Plus session where a duplicate insert is met with an ORA error number of minus 1 and an Oracle error text message. The aforementioned Java procedure includes a try-catch exception handler for the same type of primary key constraint violation. This try-catch handler uses the getErrorCode() method, which returns the five-digit ORA number—in this case, 1. Such an error may be unlikely, but it would not be impossible. Error handling code is supposed to be used infrequently, but it is not supposed to be missing from the application.

The aforementioned Java method, insertStudent(), inserts STUDENTS_PK_SEQ.NEXTVAL as the new student ID for column STUDENT_ID. This expression "NEXTVAL" always evaluates to an integer. Even though the column for STUDENT_ID is string, VARCHAR2(10), the sequence result is implicitly converted to a string by Oracle.

You can manipulate the sequence and use it to build a unique string that satisfies a desired format. We are using a VARCHAR2(10) data type for STUDENT_ID. Suppose we want a STUDENT_ID to be the letter "A" followed by a string of nine digits. We could declare a nine-digit sequence from 1 to 999,999,999—then, concatenate that string with our prefix letter. Our CREATE SEQUENCE statement would first declare the MINVALUE and MAXVALUE using these limits. INSERT statements would then "zero-left-pad" the sequence and concatenate the prefix. The INSERT statement by itself would be the following.

```
INSERT INTO students
VALUES ('A' || LPAD(student_id.NEXTVAL, 9, '0'), etc);
```

Sequence numbers are a safe strategy for guaranteeing unique values. The default CREATE SEQUENCE syntax will generate 10^{27} numbers before you cycle around—this should handle most applications.

The auto-generation feature of sequences should not be a reason to forgo error handling code. An application program using the sequence may be the main vehicle for adding students, but rare events may force inserts from other means, such as SQL*Plus. A rare situation could cause a student to be added "by hand"—possibly due to a problem with a missed student, resolved by an operations person entering the student with SQL*Plus. The operations staff may just use the next highest number in the STUDENTS table for that INSERT. The next time the application runs and enters a new student, it will generate the next sequence value and collide with what operations did with SQL*Plus.

Problems with sequences can also occur when applications are migrated and a sequence is inadvertently dropped and recreated. In this case the sequence starts over with a student ID of 1 and causes a duplicate insert. With regard to error handling, the sequence does provide uniqueness, but mistakes happen. As rare as a duplicate insert might be, a graceful capture of a primary key constraint violation will always save hours of troubleshooting.

3.1.6 Concatenated Primary Key

In this section we make use of a table, STUDENT_VEHICLES, that stores information on vehicles that students keep on campus. The table description is:

```
Name                           Null?    Type
------------------------------ -------- ------------
STATE                          NOT NULL VARCHAR2(2)
TAG_NO                         NOT NULL VARCHAR2(10)
VEHICLE_DESC                   NOT NULL VARCHAR2(30)
STUDENT_ID                     NOT NULL VARCHAR2(10)
PARKING_STICKER                NOT NULL VARCHAR2(10)
```

Your client, the motor vehicle department of the school, expresses a need, "We want to track student vehicles on campus. Students come from all over the country with cars. Students are allowed to keep cars on campus, but the college needs to track vehicles. The college issues a parking sticker that permits the car to remain on campus."

Data model analysis begins with looking at vehicle tag information. License's tags are issued by each state, so we can be certain that all vehicle license tag numbers issued by New York are unique within that state and all tag numbers from California are unique within that state. Tag numbers are short-string combinations of letters and numbers.

We can assume the following rule: within each state, all tags within that state are unique. Consequently, the combination of state abbreviation and the tag number of any student's vehicle will be always unique among all students. This rule is enforced with a combination of these two columns forming a concatenated primary key.

Below is sample data for California (CA) and New York (NY) license numbers for three students. There is no rule about a student registering more than one vehicle on campus. As you can see, student A104 has two registered vehicles.

```
STATE TAG_NO   VEHICLE_DESC  STUDENT_ID PARKING_STICKER
----- -------- ------------  ---------- ---------------
CA    CD 2348  1977 Mustang  A103       C-101-AB-1
NY    MH 8709  1989 GTI      A104       C-101-AB-2
NY    JR 9837  1981 Civic    A104       C-101-AB-3
```

We will store this information in a new table called STUDENT_VEHICLES. We know there will be other columns of interest (e.g., vehicle registration information); for now, we'll just include vehicle description, the student who uses the car, and the number of the parking sticker issued by the campus police. The key here is that, in the real world, we can definitely say that the combination of a state abbreviation and license tag number is always unique—this makes (STATE, TAG_NO) a concatenated primary key. The DDL for this table is:

```
CREATE TABLE student_vehicles
 (state           VARCHAR2(2),
  tag_no          VARCHAR2(10),
  vehicle_desc    VARCHAR2(20),
  student_id      VARCHAR2(10),
  parking_sticker VARCHAR2(10)) TABLESPACE student_data;

ALTER TABLE student_vehicles
  ADD CONSTRAINT pk_student_vehicles
  PRIMARY KEY (state, tag_no)
  USING INDEX TABLESPACE student_index;
```

Concatenated primary keys and sequence-generated keys do not mix. A single column is all that is needed for a sequence-generated primary key. This table should not incorporate a sequence; it stores information on vehicles that naturally and uniquely distinguishes each vehicle by state and tag number.

During your initial design, you may be told by your client that certain values will always be unique. Your client may say, for example with an inventory system, that the combination of COMPANY and PART_NO will always be unique. You go with this concept and construct a concatenated primary key. Months later, you learn that there are exceptions—sometimes this is as much a revelation to the client as it is to you. Your approach then is to rebuild the tables and add a new column for the sequence.

Some modelers will, as a standard practice, make every primary key a sequence-generated column. This works but, to a degree, discards reality. For example, there is no doubt that a state abbreviation and a tag number, in the real world, are unique and truly can be used to uniquely identify the attributes of any vehicle on campus.

When the de facto standard in a project is to make all primary keys sequence-generated, you have to pay closer attention to how you query the table because you may frequently query the table using real-world attributes like state abbreviation and tag number. These columns might not have indexes, unless you specifically create them. The column with the sequence has an index because it has a primary key; however, other columns may not have, but should have indexes.

3.1.7 Extra Indexes with Pseudo Keys

The STUDENT_VEHICLES table has a natural primary key—it seems natural to assume that the combination of state abbreviation and license number will always be unique. Suppose we add a pseudo key, insisting on a new column called VEHICLE ID. We'll make VEHICLE_ID the primary key. We then have the following table.

```
CREATE TABLE student_vehicles
 (vehicle_id      NUMBER,
  state           VARCHAR2(2),
  tag_no          VARCHAR2(10),
  vehicle_desc    VARCHAR2(20),
  student_id      VARCHAR2(10),
  parking_sticker VARCHAR2(10)) TABLESPACE student_data;

ALTER TABLE student_vehicles
  ADD CONSTRAINT pk_student_vehicles
  PRIMARY KEY (vehicle_id)
  USING INDEX TABLESPACE student_index;

CREATE SEQUENCE student_vehicles_pk_seq;
```

Suppose we have the same sample data as Section 3.1.6, "Concatenated Primary Key." The primary key, VEHICLE_ID, is a sequence number that starts with 1. After adding three vehicles we have the following data.

```
VEH_ID STATE TAG_NO   VEHICLE_DESC  STUDENT_ID PARKING_STICKER
------ ----- -------- ------------  ---------- ---------------
     1 CA    CD 2348  1977 Mustang  A103       C-101-AB-1
     2 NY    MH 8709  1989 GTI      A104       C-101-AB-2
     3 NY    JR 9837  1981 Civic    A104       C-101-AB-3
```

The following SQL queries the table using the primary key column in the query. This query will use the index.

```
SELECT * FROM student_vehicles WHERE vehicle_id = '1';
```

The execution plan, shown here, will include the index.

```
Execution Plan
---------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE
1  0   TABLE ACCESS (BY INDEX ROWID) OF 'STUDENT_VEHICLES'
2  1     INDEX (UNIQUE SCAN) OF 'PK_STUDENT_VEHICLES'
         (UNIQUE)
```

What if the application code relies on the columns for state and license number? This is a likely possibility. A lookup of information based on a state and license plate number could be dependent on fields selected on an HTML form. The application developer could have used the primary key, but chose to use the other columns. The following SQL will return the same information as the previous select statement.

```
SELECT *
  FROM students
 WHERE state = 'CA'
   AND tag_no = 'CD 2348';
```

The difference is that the first query will likely run faster. If there are a significant number of students, the first query will use the index of the primary key. The second query will not use an index because there is no index on columns STATE and TAG_NO. The execution plan shows a table scan that will cause considerable wait time for the end user.

```
Execution Plan
----------------------------------------------------------
0    SELECT STATEMENT Optimizer=CHOOSE
1    0   TABLE ACCESS (FULL) OF 'STUDENT_VEHICLES'
```

When a pseudo key is created, as earlier, and there is a natural primary key in the table, there is the risk that the application developed will take fields, such as STATE and TAG NO from a form (i.e., an HTML form or client GUI form) and use those fields to query the database. When this happens, the performance of that query is slow unless actions are taken to identify these table scans and resolve them with additional indexes.

The decision to add a pseudo key is not being judged here—the issue is to recognize when and if the primary key, whatever that key may be, is being used and when it is not being used. There are three options when the application is not using the sequence primary key.

1. Don't use the pseudo index, VEHICLE_ID. It provides no benefit because there is a natural primary key built into the data: STATE and TAG_NO.
2. Create an index on the columns STATE and TAG_NO.
3. Create a concatenated UNIQUE constraint on columns STATE and TAG_NO; the mere creation of this unique constraint will enforce the natural business rule that these columns are unique, plus create an index on these columns.

Why would someone choose to use a sequence rather than columns that, in the real world, identify a unique row? This is a decision made by the data modeler. Maybe the natural primary key is many columns—financial applications that deal with financial instruments and trading of those instruments have so many attributes that the natural primary key of a table is many columns. That doesn't mean those columns can't be a primary key. An excessive number of columns in a primary key means large indexes and possible foreign keys that, if we have indexes on those, means larger indexes. Many columns in the primary key may not be desirable.

Secondly, maybe the modeler has doubts as to whether, in the future, those columns will remain unique. Sometimes end users say the data is unique; the data modeler creates a primary key only to find out later that the real data does, in fact, contain duplicates. So, reverting to a concatenated primary key may not be the first choice.

Suppose you stick with the pseudo key, such as VEHICLE_ID. Given this approach, we can still consider the business rule that, within the STUDENT_VEHICLES table, the STATE and TAG_NO are still unique and you can construct a UNIQUE constraint on those columns. The UNIQUE constraint does generate an index and consequently the query following will have the potential to perform at optimal levels with the use of the index generated through the unique constraint.

```
SELECT *
  FROM student_vehicles
 WHERE state = 'CA'
   AND tag_no = 'CD 2348';
```

The DDL for the scenario to implement the pseudo sequence plus a concatenated unique constraint on STATE and TAG_NO is the following.

```
CREATE TABLE student_vehicles
 (vehicle_id      NUMBER,
  state           VARCHAR2(2),
  tag_no          VARCHAR2(10),
  vehicle_desc    VARCHAR2(20),
  student_id      VARCHAR2(10),
  parking_sticker VARCHAR2(10)) TABLESPACE student_data;

ALTER TABLE student_vehicles
  ADD CONSTRAINT pk_student_vehicles
  PRIMARY KEY (vehicle_id)
  USING INDEX TABLESPACE student_index;

CREATE SEQUENCE student_vehicles_pk_seq;

ALTER TABLE student_vehicles
  ADD CONSTRAINT uk_student_vehicles_state_tag
  UNIQUE (state, tag_no)
  USING INDEX TABLESPACE student_index;
```

If you do not expect state and license number to be or remain unique, you can always create a nonunique index on these columns. So, rather than add a unique constraint as earlier, replace that with the following, which only creates an index.

```
CREATE INDEX student_vehicles_state_tag
  ON student_license(state, tag_no)
  TABLESPACE student_index;
```

In summary, when your approach to primary keys is, by default, to always use sequence-generated primary keys, consider how you query the table. Are there attributes within the table to which a unique constraint can be applied? If so, create a unique constraint. For all other columns, look at how the table is accessed and add additional indexes to optimize performance.

3.1.8 Enable, Disable, and Drop

You can drop a primary key constraint with the following.

```
ALTER TABLE <table_name> DROP CONSTRAINT
<constraint_name>;
```

This drops the constraint and drops the index associated with that constraint.

Oracle will not allow you to drop a primary key to which there is a referencing foreign key. If a table has a referencing foreign key you will get this error.

```
ORA-02273: this unique/primary key is referenced
by some foreign keys
```

You can DROP CASCADE the constraint. This will drop the primary key constraint and all foreign keys that reference that parent. This does not require that the foreign key constraint be declared with the CASCADE option. You can always drop a primary key with the CASCADE option, but it is permanent. The primary key and foreign key constraints are deleted from the data dictionary. The CASCADE option is:

```
ALTER TABLE state_lookup
DROP CONSTRAINT state_lookup CASCADE;
```

You can check all referencing foreign keys to a primary key constraint with the script MY_CHILDREN_ARE in Chapter 5, Section 5.6.5.

A less drastic measure is to disable a constraint. The CASCADE restriction applies as well to disabling constraints. If there are referencing foreign keys, you can disable a primary key constraint with:

```
ALTER TABLE state_lookup
DISABLE CONSTRAINT state_lookup CASCADE;
```

A disabled constraint is still defined in the data dictionary, it is just not being enforced. Furthermore, the status is set to DISABLED. With the CASCADE option, the status of the primary key and all referencing foreign key constraints are set to a DISABLED status.

The index, after a primary key constraint is disabled, is gone—deleted from the data dictionary; however, it is immediately rebuilt when the constraint is enabled.

```
ALTER TABLE state_lookup ENABLE CONSTRAINT
pk_state_lookup;
```

This ALTER TABLE statement recreates the index and sets the primary key constraint to ENABLED. The foreign key constraints are still disabled. Each of these has to be enabled with:

```
ALTER TABLE students ENABLE CONSTRAINT fk_students_state;
```

An index on a primary key column cannot be dropped—it would be dropped with a DROP or DISABLE constraint. Indexes can be rebuilt with no affect on the constraint with:

```
ALTER INDEX pk_state_lookup REBUILD;
```

The ENABLE and DISABLE keywords can be appended to any constraint declaration. Attaching DISABLE creates the constraint and sets the state to DISABLED. It can be enabled any time. The ENABLE keyword is the default for all constraint clauses.

3.1.9 Deferrable Option

A primary key constraint can be created with the option DEFERRABLE. This option permits an application program to disable the constraint during a load. The assumption is that manipulation of the table will occur, but when the load is complete the data will conform to the rules of the primary key constraint. The application will COMMIT the loaded data. At that COMMIT, the constraint is enforced. Should the program have left invalid data in the table—data that violates the constraint—the transaction is rolled back.

Scenario 1 Bad Data Is Loaded with Deferrable

A stored procedure loads 1,000 rows. There are duplicates. All 1,000 rows are loaded. The program does a commit. The transaction is rolled back because the data violates the constraint.

Scenario 2 Bad Data Is Loaded without Deferrable

A stored procedure loads 1000 rows. There are duplicates in this data. The program proceeds to load the data and at some point a duplicate is inserted. The insert fails. The application can choose to ignore this single insert (using exception handling code) and continue with the remaining data. The program is also capable of rolling back the transaction.

The DEFERRABLE option is created with the following syntax—a sample table PARENT is created with two columns.

```
CREATE TABLE parent
(parent_id   NUMBER(2),
 parent_desc VARCHAR2(10));

ALTER TABLE parent ADD CONSTRAINT pk_parent PRIMARY KEY
(parent_id) DEFERRABLE;
```

The aforementioned constraint definition is the same as this next statement:

```
ALTER TABLE parent ADD CONSTRAINT pk_parent PRIMARY KEY
(parent_id) DEFERRABLE INITIALLY DEFERRED;
```

Scenario 1 is shown with the following code. This PL/SQL block loads duplicate data. After the load, but before the COMMIT, a duplicate is DELETED. This block completes with success because the duplicate was removed prior to the commit.

```
DECLARE
BEGIN
   EXECUTE IMMEDIATE 'SET CONSTRAINTS ALL DEFERRED';

   INSERT INTO parent values (1,'A');
   INSERT INTO parent values (1,'B');
   INSERT INTO parent values (3,'C');
   INSERT INTO parent values (4,'D');

   DELETE FROM parent WHERE parent_desc = 'B';
   COMMIT;
END;
```

If this block had not removed the duplicate, an Oracle error would occur. That error is the same error as a primary key constraint violation. The following block accomplishes the same task, loading the same data. This scenario does not use the deferrable option. Rather, all good data remains in the PARENT table.

```
BEGIN
    BEGIN
        INSERT INTO parent values (1, 'A');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;
    BEGIN
        INSERT INTO parent values (1, 'B');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;
    BEGIN
        INSERT INTO parent values (3, 'C');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;
    BEGIN
        INSERT INTO parent values (4, 'D');
    EXCEPTION WHEN DUP_VAL_ON_INDEX THEN null;
    END;

    COMMIT;
END;
```

The data remaining in the table is the first, third, and fourth row. The DEFERRABLE option loads everything. Then the constraint is applied and the transaction is committed or rolled back. The second scenario provides the option to roll back a duplicate insert. Alternatively, the application can skip the duplicate insert and continue processing good data.

The deferrable option should not be used to permit the table to be a work area. Scenario 1 seems to permit the application to load data, and then manipulate that data, cleaning out bogus records. Hopefully, the data will be good upon a commit. Alternatives exist. Temporary tables can be created for loading data for manipulation. After manipulation and cleansing of the data, the rows are inserted into the production table. Temporary tables can persist for the life of a session or transaction. The following DDL creates a temporary table that can be used as a private in-memory table that persists for the duration of a transaction.

```
CREATE GLOBAL TEMPORARY TABLE
parent_temp
(parent_id   NUMBER(2),
 parent_desc VARCHAR2(10)) ON COMMIT DELETE ROWS;
```

Scenario 1 can now use the temporary table for massaging the raw data. When data is moved into the permanent table, the commit deletes rows from the temporary table. The temporary table is private to this transaction. The following PL/SQL block is Scenario 1 using a temporary table with no deferrable option.

```
BEGIN
    INSERT INTO parent_temp values (1,'A');
    INSERT INTO parent_temp values (1,'B');
    INSERT INTO parent_temp values (3,'C');
    INSERT INTO parent_temp values (4,'D');

    DELETE FROM parent_temp WHERE parent_desc = 'B';

    INSERT INTO parent SELECT * FROM parent_temp;
    COMMIT;
END;
```

The DEFERRABLE option enforces the constraint with each DML statement. This is the default behavior of the constraint without the DEFERRABLE option; however, the DEFERRABLE does provide for the following:

- An application can SET CONSTRAINTS ALL DEFERRED, load data, and have the constraint enforced when the transaction completes with a COMMIT statement.
- The constraint can be disabled with an ALTER TABLE statement to DISABLE the constraint. Data can be loaded. The constraint can be enabled with NOVALIDATE, leaving duplicates in the table but enforcing the constraint for future DML (see Section 3.1.10, "NOVALIDATE.")

The DEFERRABLE option can be declared with the following attribute:

```
DEFERRABLE INITIALLY DEFERRED
```

The INITIALLY DEFERRED feature defaults to enforcing the constraint only when a transaction commits. When this is set, each DML statement is not individually enforced. Enforcement takes place only when the transaction completes. This option provides the following:

- An application begins a transaction knowing that the constraint is enforced upon the commit. The application loads data, then commits. The entire transaction is accepted or rejected. The code contains INSERT statements and a COMMIT. There is nothing in the code to reflect that the INSERTS are validated only upon commit. Prior to a commit, the application can specifically check for validation with the following.

  ```
  EXECUTE IMMEDIATE 'SET CONSTRAINTS ALL IMMEDIATE';
  ```

The following summarizes DEFERRABLE option.

1. You can declare the constraint DEFERRABLE, which has the following attributes in the data dictionary USER_CONSTRAINTS view.

   DEFERRABLE = DEFERRABLE

   DEFERRED = IMMEDIATE

   This option means that the constraint is enforced with each DML. You can write code, as shown earlier, that directs Oracle to refrain from constraint enforcement until a COMMIT or a ROLLBACK. This requires a SET CONSTRAINTS statement in your code.

2. You can declare the constraint with the option DEFERRABLE INITIALLY DEFERRED. This sets the constraint in the data dictionary to:

   DEFERRABLE = DEFERRABLE

   DEFERRED = DEFERRED

3. This option means that the default behavior for a transaction is to enforce constraints when the transaction does a COMMIT or ROLLBACK. A PL/SQL program would look like any other program with this option. One would have to look into the status of the constraint to see when the constraint is enforced. An application can choose to enforce constraints with the SET CONSTRAINTS ALL IMMEDIATE command. Without this statement, the constraint will be enforced when the transaction completes.

4. You can ALTER the DEFERRED status of a constraint. If you declare it as DEFERRABLE, then it has the status of:

   DEFERRABLE = DEFERRABLE

   DEFERRED = IMMEDIATE

   You can execute the following:

   ```
   ALTER TABLE table_name MODIFY CONSTRAINT
   constraint_name INITIALLY DEFERRED.
   ```

   This changes the state to:

   DEFERRABLE = DEFERRABLE

   DEFERRED = DEFERRED

5. You can ALTER the DEFERRED status of a constraint declared INITIALLY DEFERED with the following:

   ```
   ALTER TABLE table_name MODIFY CONSTRAINT
   constraint_name INITIALLY IMMEDIATE.
   ```

   This changes the state to:

   DEFERRABLE = DEFERRABLE

   DEFERRED = IMMEDIATE

As always, you can disable the constraint and then enable it with ALTER TABLE commands.

3.1.10 NOVALIDATE

The NOVALIDATE option allows nonconforming data to be loaded and left in the table while the rule of the constraint is enabled only for future inserts. This option can be used in data warehouse systems where management must have historical data for analysis. Historical data will frequently violate present day business rules.

To load noncompliant data, the constraint must be initially created with the deferrable option. Prior to loading historical data, the constraint must be disabled. The following creates a table with a deferrable primary key constraint. Prior to the load, the constraint is disabled. Afterward, the constraint is enabled with the NOVALIDATE option. From this point forward, the historical data remains but all new inserts will be constrained to the rules of the primary key.

```
CREATE TABLE parent
(parent_id   NUMBER(2),
 parent_desc VARCHAR2(10));

ALTER TABLE parent ADD CONSTRAINT pk_parent PRIMARY KEY
(parent_id) DEFERRABLE;

ALTER TABLE parent DISABLE CONSTRAINT pk_parent;

BEGIN
    INSERT INTO parent values (1,'A');
    INSERT INTO parent values (1,'B');
    INSERT INTO parent values (3,'C');
    INSERT INTO parent values (4,'D');
END;

ALTER TABLE parent ENABLE NOVALIDATE CONSTRAINT pk_parent;
```

After the aforementioned PL/SQL block executes, duplicates exist in the tables; however, all new inserts must conform to the primary key.

3.1.11 Error Handling in PL/SQL

A duplicate insert in PL/SQL is easily captured with the PL/SQL built-in exception. The exception name is

```
DUP_VAL_ON_INDEX
```

Including an exception handler will allow an application to handle the rare case of a duplicate. The following stored procedure returns a Boolean, indicating a failure if the insert is a duplicate.

```
CREATE OR REPLACE
FUNCTION insert_parent
    (v_id NUMBER, v_desc VARCHAR2) RETURN BOOLEAN
IS
BEGIN
    INSERT INTO parent VALUES (v_id, v_desc);
    return TRUE;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN return FALSE;
END;
```

3.2 UNIQUE

In this section we use a table that stores student information. The table description is:

```
Name                          Null?    Type
------------------------------ -------- ------------
STUDENT_ID                             VARCHAR2(10)
STUDENT_NAME                           VARCHAR2(30)
COLLEGE_MAJOR                          VARCHAR2(15)
STATUS                                 VARCHAR2(15)
STATE                                  VARCHAR2(2)
LICENSE_NO                             VARCHAR2(30)
```

The UNIQUE constraint is applied to a column, or set of columns, to enforce the following rule: If a value exists, than that value must be unique.

This definition sounds similar to a PRIMARY KEY constraint. The following is a comparison between the primary key and unique constraints.

- A table can have just one primary key constraint but a table can have more than one unique constraint.
- A column that is part of a primary key can never have a NULL value. A column that is part of a unique constraint can be null. If a column has a unique constraint, there can be many rows with a NULL for that column. The values that are not null must be unique.
- When we create a primary key we create or reuse an index. The same is true for UNIQUE constraints.
- The primary key and unique constraint columns can be the parent in a foreign key relationship. The columns of a foreign key constraint frequently refer back to columns of a primary key constraint. They can also refer back to columns of a unique constraint.

The following DDL creates a concatenated unique constraint on columns (A, B):

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER, b NUMBER);
ALTER TABLE temp ADD CONSTRAINT uk_temp_a_b UNIQUE (a, b);
```

NULLs are permitted in any column provided the data that does exist qualifies as unique. The following INSERT statements are valid.

```
-- UNIQUE Constraint on last 2 columns.
INSERT INTO temp VALUES (1,    1,    1);
INSERT INTO temp VALUES (2,    2,    1);
INSERT INTO temp VALUES (3,    1,    2);
INSERT INTO temp VALUES (4, NULL, NULL);
INSERT INTO temp VALUES (5, NULL, NULL);
INSERT INTO temp VALUES (6,    1, NULL);
INSERT INTO temp VALUES (7, NULL,    1);
INSERT INTO temp VALUES (8,    2, NULL);
```

The following duplicates the last insert and raises a constraint violation.

```
SQL> insert into temp values (9, 2, NULL);
insert into temp values (9, 2, NULL)
*
ORA-00001: unique constraint (SCOTT.UK_TEMP_A_B) violated
```

Notice that the prefix "ORA" and a minus 1 for the error code are identical to the primary key constraint violation.

3.2.1 Combining NOT NULL, CHECK with UNIQUE Constraints

A NOT NULL constraint is sometimes added to the UNIQUE constraint. This additional requirement stipulates that the column values must be unique and that NULLs are not allowed. The following illustrates this case.

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER NOT NULL);
ALTER TABLE temp ADD CONSTRAINT uk_temp_a UNIQUE (a);
```

Concatenated unique constraints are often supplemented with a CHECK constraint that prevents the condition: one column is NULL and one column has a value. This is enforced following this example.

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER, b NUMBER);
ALTER TABLE temp ADD CONSTRAINT uk_temp_a_b UNIQUE (a, b);
ALTER TABLE temp ADD CONSTRAINT ck_temp_a_b CHECK
    ((a IS NULL AND B IS NULL) OR
     (a IS NOT NULL AND b IS NOT NULL));
```

The aforementioned CHECK and UNIQUE constraint combination allows the first two inserts, but rejects the second two inserts.

```
INSERT INTO temp VALUES (6,    1,    1);  -- successful
INSERT INTO temp VALUES (7, NULL, NULL);  -- successful
INSERT INTO temp VALUES (6,    1, NULL);  -- fails
INSERT INTO temp VALUES (7, NULL,    1);  -- fails
```

Combining NOT NULL and CHECK constraints with UNIQUE constraints allows for several options.

- The column values, when data is present, must be unique. This is enforced with the UNIQUE constraint.
- Any column or all columns in the UNIQUE constraint can be mandatory with NOT NULL constraints on individual columns.
- Combinations of NULL and NOT NULL restrictions can be applied using a CHECK constraint and Boolean logic to dictate the rule.

3.2.2 Students Table Example

Consider a table, STUDENTS, that includes columns for student driver's license information. Not every student has a license—this has to be considered. Our rules are:

- Every student must have a unique identification number or, STUDENT_ID—this is the primary key.
- If a student has a driver's license the concatenation of license number and state abbreviation must be unique. The columns for state and license are not mandatory and do not have a NOT NULL constraint.
- The final CHECK constraint ensures that should a student have a license, the state and license values are both entered into the system.

The following DDL creates the STUDENTS table with a concatenated UNIQUE constraint on STATE and LICENSE_NO.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(15) NOT NULL,
  state         VARCHAR2(2),
  license_no    VARCHAR2(30)) TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT uk_students_license
  UNIQUE (state, license_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT ck_students_st_lic
   CHECK ((state IS NULL AND license_no IS NULL) OR
(state IS NOT NULL AND license_no is NOT NULL));
```

The following paragraphs summarize key points about the above DDL.

- We call the primary key constraint PRIMARY KEY. The unique constraint is just "UNIQUE." In conversation we often refer to "unique key" constraints, but when writing DDL, leave off the "KEY" part. Also, unique constraints are often named with a prefix of "UK." The primary key, foreign key, and unique constraint all work together to enforce referential integrity. But the DDL syntax for a unique constraint does not include the "KEY" keyword.

- The unique constraint causes an index to be created; therefore, we have included the tablespace as the location for this index (if an index on these columns has already been created, then the constraint will use that index). For all the DDL in this text, the tables are created in a STUDENTS_DATA tablespace and all indexes are created in a STUDENTS_INDEX tablespace—a fairly standard practice.

- The unique constraint is named UK_STUDENTS_LICENSE. Constraint names are limited to 30 characters. Constraint names in this text are preceded with a prefix to indicate the constraint type. This is followed by the table name. For a primary key constraint, that is all we need. For other constraints, we try to append the column name—this can be difficult because table names and column names may be up to 30 characters. Sometimes you must abbreviate. Most important, the name of the constraint should clearly indicate the constraint type and table. The column names included in the constraint can be short abbreviations—this approach helps when resolving an application constraint violation.

3.2.3 Deferrable and NOVALIDATE Options

Similar to primary key constraints, a UNIQUE constraint can be declared as deferrable. The constraint can be disabled and enabled with ALTER TABLE statements. All the options described in Section 3.1.9, "Deferrable Option," and 3.1.10, "NOVALIDATE," are applicable to UNIQUE constraints.

3.2.4 Error Handling in PL/SQL

A duplicate insert, for a primary key and unique constraint, is captured with the PL/SQL built-in exception. The exception name is

```
DUP_VAL_ON_INDEX
```

The following procedure inserts a student and captures unique constraint errors. The code for a check constraint error is also captured. The check constraint error number is mapped to a declared exception. The procedure then includes a handler for that exception. The primary key and unique constraint errors have the predefined exception declared in the language. Other types of errors, such as check constraint errors, need exceptions declared that must be mapped to the Oracle error number. The check constraint error number is minus 2290.

```
 CREATE OR REPLACE PROCEDURE
    insert_student(v_student_id VARCHAR2,
        v_student_name VARCHAR2,
        v_college_major VARCHAR2,
        v_status VARCHAR2,
        v_state VARCHAR2,
        v_license_no VARCHAR2)
IS
    check_constraint_violation exception;
    pragma exception_init(check_constraint_violation, -2290);
BEGIN
    INSERT INTO students VALUES
        (v_student_id, v_student_name,
         v_college_major, v_status,
         v_state, v_license_no);

    dbms_output.put_line('insert complete');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('PK or unique const violation');
    WHEN check_constraint_violation THEN
        dbms_output.put_line('check constraint violation');
END;
```

To test the error handling logic, use SQL*Plus to EXECUTE the procedure with some data. The first two inserts fail because the state and license violate the check constraint rule: both are NULL or both are NOT NULL.

The third and fourth inserts work. The last insert fails because it violates the unique constraint, which is a duplicate of the prior insert.

```
insert_student('A900','Ann','Math','Degree','CA',NULL);
check constraint violation

insert_student('A900','Ann','Math','Degree',NULL,'ABC');
check constraint violation

insert_student('A900','Ann','Math','Degree',NULL,NULL);
insert complete

insert_student('A902','Joe','Math','Degree','CA','ABC');
insert complete

insert_student('A903','Ben','Math','Degree','CA','ABC');
PK or unique const violation
```

3.3 Foreign Key

Foreign key constraints enforce referential integrity. A foreign key constraint restricts the domain of a column value. An example is to restrict a STATE abbreviation to a limited set of values in another control structure—that being a parent table.

The term "lookup" is often used when referring to tables that provide this type of reference information. In some applications, these tables are created with this keyword—a practice we'll use here with the example STATE_LOOKUP.

Start with creating a lookup table that provides a complete list of state abbreviations. Then use referential integrity to ensure that students have valid state abbreviations. The first table is the state lookup table with STATE as the primary key.

```
CREATE TABLE state_lookup
 (state      VARCHAR2(2),
  state_desc VARCHAR2(30)) TABLESPACE student_data;

ALTER TABLE state_lookup
  ADD CONSTRAINT pk_state_lookup PRIMARY KEY (state)
  USING INDEX TABLESPACE student_index;
```

To insert a few rows:

```
INSERT INTO state_lookup VALUES ('CA', 'California');
INSERT INTO state_lookup VALUES ('NY', 'New York');
INSERT INTO state_lookup VALUES ('NC', 'North Carolina');
```

We enforce referential integrity by implementing the parent–child relationship, graphically shown in Figure 3-2.

Figure 3-2. Foreign Key with State Lookup.

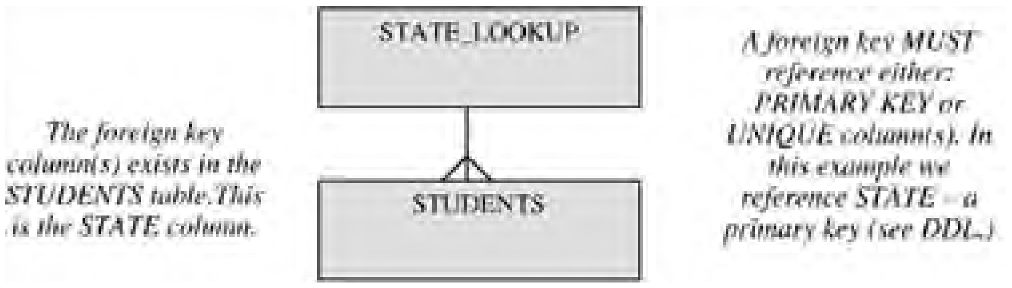

The foreign key column(s) exists in the STUDENTS table. This is the STATE column.
STATE_LOOKUP
STUDENTS
A foreign key MUST reference either: PRIMARY KEY or UNIQUE column(s). In this example we reference STATE – a primary key (see DDL.)

Figure 3-2 shows a one-to-many relationship between the STATE_LOOKUP table and the STUDENTS table. The STATE_LOOKUP table defines the "universal set" of state abbreviations—each state being represented once in that table; hence, a primary key on the STATE column of STATE_LOOKUP.

A state from the STATE_LOOKUP table can appear multiple times in the STUDENTS table. There can be many students from a single state. Hence referential integrity implements a one-to-many relationship between STATE_LOOKUP and STUDENTS.

The foreign key also ensures the integrity of the STATE column in the STUDENTS table—a student with a driver's license will always have a state abbreviation that is a member of the STATE_LOOKUP table.

The foreign key constraint is created on the child table. Following is the STUDENTS table with a foreign key constraint. With an ALTER TABLE statement, we declare the column STATE to have a foreign key constraint that references the primary key column of the STATE_LOOKUP table.

Script 3-1 Foreign Key with State Lookup DDL.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(20) NOT NULL,
  state         VARCHAR2(2),
  license_no    VARCHAR2(30)) TABLESPACE student_data;
```

```
ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT uk_students_license
  UNIQUE (state, license_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT ck_students_st_lic
   CHECK ((state IS NULL AND license_no IS NULL) OR
          (state IS NOT NULL AND license_no is NOT NULL));

ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state);
```

The DDL script in Script 3-1 creates the STUDENTS table and table constraints. These constraints enforce the following rules:

Rule	Enforced With
A student is uniquely identified by a STUDENT_ID.	PRIMARY KEY constraint.
A student MAY have a driver's license. If they do, that state and license combination is unique among all other students.	UNIQUE constraint on STATE and LICENSE.
If STATE is NULL then LICENSE_NO must be NULL; otherwise both must be NOT NULL.	CHECK constraint on STATE and LICENSE.
A student MAY have a column value for STATE. If they do, the STATE abbreviation is valid with respect to the STATE_LOOKUP table.	FOREIGN KEY constraint on STATE.

3.3.1 Four Types of Errors

The rules of referential integrity are enforced during updates and deletes on parent tables and inserts and updates on child tables. The SQL statements affected by referential integrity are:

PARENT-UPDATE	You cannot UPDATE a STATE in STATE_LOOKUP with a value so as to leave students with a state abbreviation that is no longer in the STATE_LOOKUP table.
PARENT-DELETE	You cannot DELETE a STATE and leave students with a state that is no longer in the parent lookup table. For example, if there are students with a California license, which use the abbreviation 'CA,' you cannot delete the 'CA' row from STATE_LOOKUP.
CHILD-INSERT	You cannot INSERT a student with a state that is not found in the STATE_LOOKUP table. For example, you cannot insert a student with a license and set the STATE column to a value not found in the STATE_LOOKUP table.
CHILD-UPDATE	You cannot UPDATE a student and replace the state with a state not found in the parent state lookup table.

The following SQL statements demonstrate the four error types and the Oracle error returned when the constraint is violated. These inserts, updates, and deletes behave assuming the data for the STATE_LOOKUP and STUDENTS tables:

STATE_LOOKUP

State	State Description
CA	California
NY	New York
NC	North Carolina

STUDENTS

Student ID	Student Name	College Major	Status	State	License NO
A101	John	Biology	Degree	NULL	NULL
A102	Mary	Math/Science	Degree	NULL	NULL

A103	Kathryn	History	Degree	CA	MV-232-13
A104	Steven	Biology	Degree	NY	MV-232-14
A105	William	English	Degree	NC	MV-232-15

The first two SQL statements are changes to the parent table. Prior to changing the parent, Oracle must examine the contents of the child table to ensure data integrity.

PARENT-UPDATE

```
SQL> UPDATE state_lookup
  2  SET state = 'XX'
  3  WHERE state = 'CA';

UPDATE state_lookup
*
ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated – child record found
```

PARENT-DELETE

```
SQL> DELETE FROM state_lookup
     2  WHERE state = 'CA';

DELETE FROM state_lookup
*
ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated – child record found
```

The next two statements are changes to the child table. Each DML on the child requires that Oracle examine the contents of the parent to ensure data integrity.

CHILD-INSERT

```
SQL> INSERT INTO STUDENTS
  2  VALUES ('A000',
  3  'Joseph','History','Degree','XX','MV-232-00');

INSERT INTO STUDENTS
*
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated - parent key not found
```

CHILD-UPDATE

```
SQL> UPDATE students
  2  SET state = 'XX'
  3  WHERE student_id = 'A103';

UPDATE students
*
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.FK_STUDENTS_STATE)
violated - parent key not found
```

For Oracle to enforce these four rules, it must read from both tables. A simple INSERT into a STUDENTS table, given a foreign key constraint, requires that Oracle read the STATE_LOOKUP table. If we DELETE from the STATE_LOOKUP table, then Oracle must read the STUDENTS table to first ensure there is no student row that contains a STATE value that references the STATE_LOOKUP row.

In each of the four types of errors above, the error number is the same: ORA-02291.

Referential integrity is a critical part of a database design. It is rare for a table to not be either a parent or child of some other table. If you ever look at a data model printed on a large scale graphics plotter, the first thing you will notice are tables with no lines—no parents and no children.

3.3.2 Delete Cascade

You have the option within the foreign key syntax to specify a delete cascade feature. This feature only affects delete statements in the parent table.

With this option, a delete from the parent will automatically delete all relevant children. Had we created the foreign key constraint in Script 3-1, "Foreign Key," with the DELETE CASCADE option, then the following SQL would delete the record in the STATE_LOOKUP table for California plus all students that have a California license.

```
SQL> DELETE FROM state_lookup
  2  WHERE state = 'CA';
```

The DELETE CASCADE syntax is:

```
ON DELETE CASCADE
```

The syntax for the foreign constraint, shown in Script 3-1, can be rewritten with the cascade option as follows:

```
ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state)
  ON DELETE CASCADE;
```

The delete cascade option should be the exception rather than the rule. Lots of data can be inadvertently lost with an accidental delete of a row in the parent lookup table. There are applications where this option is very useful. If data is temporary—only lives for a short time and is eventually deleted—then this is very convenient.

Delete cascade can span multiple tables. A parent can cascade to a child and that child can cause a delete cascade to other tables. If there is a foreign key constraint in the chain, without the cascade option, the delete from the parent fails.

Deletes that cascade over several tables can potentially affect other parts of the system. A lengthy delete that spans tables and deletes millions of rows will require comparable rollback space to write undo information. Rollback space should be analyzed if the cascade delete is excessive. Additionally, performance of concurrent queries against the tables will be affected. Consider the following when declaring a delete cascade option.

- Does the cascade fit the application? An accidental delete from a parent look table should not delete customer accounts.
- What is the chain being declared? Look at what tables cascade to other tables. Consider the potential impact and magnitude of a delete and how it would impact performance.

3.3.3 Mandatory Foreign Key Columns

The foreign key constraints stipulates the rule that a child MAY be a member of the parent. If the child is a member then there must be integrity. A NOT NULL constraint on the foreign key replaces MAY with MUST.

The foreign key constraint in Script 3-1 enforces the rule.

- A student MAY have a driver's license. If they do, that state is a member of STATE_LOOKUP.

This rule can have a different flavor—restated here:

- A student MUST have a driver's license and the state is a member of STATE_LOOKUP.

The latter is still a statement of referential integrity; there still exists a one-to-many relationship. The rule changes to MUST when the NOT NULL constraint is declared on the child column. The business rules are revised. Because the STATE and LICENSE are mandatory, the CHECK constraint is removed. The foreign key relationship changes from MAY to MUST.

Script 3-2 DDL for the STUDENTS and STATE_LOOKUP tables enforces the following:

Rule	Enforced With
A student is uniquely identified by a STUDENT_ID.	PRIMARY KEY constraint
A student MUST have a driver's license. The state and license combination is unique among all other students.	UNIQUE constraint, NOT NULL on STATE, NOT NULL on LICENSE
A student MUST have a column value for STATE. The STATE abbreviation is valid with respect to the STATE_LOOKUP table.	FOREIGN KEY constraint, NOT NULL on STATE

The DDL to enforce these rules is similar to Script 3-1. There is no CHECK constraint and NOT NULL constraints are added. There is no change to the STATE_LOOKUP table to accommodate the rule changes.

Script 3-2 Foreign Key with Mandatory Constraints.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(20) NOT NULL,
  state         VARCHAR2(2)  NOT NULL,
  license_no    VARCHAR2(30) NOT NULL)
TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT uk_students_license
  UNIQUE (state, license_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state);
```

3.3.4 Referencing the Parent Syntax

When you create a foreign key constraint, the column(s) on that foreign key reference the column(s) that make up a PRIMARY KEY or UNIQUE constraint. In the case where a foreign key references a parent primary key, the column does not need to be specified. For example, Script 3-2 uses the following syntax:

```
ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup (state);
```

When no parent column is referenced, the default referenced column is the primary key of the parent. The following would work just as well.

```
ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup;
```

When your foreign key references a parent column with a unique constraint, that parent column must be specified in the ADD CONSTRAINT statement.

3.3.5 Referential Integrity across Schemas and Databases

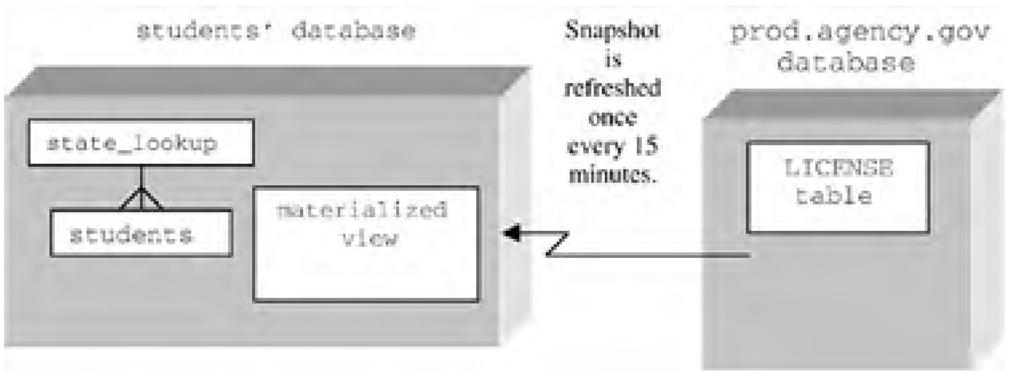
A foreign key in a table can refer to a parent table in another schema. This is not desirable, at least from a database administrator's perspective. An application that sits wholly within a single schema is very portable. The entire application that exports as a single owner easily imports into another database. When functionality spans schemas, the migration is not such a smooth process. Often, the application is not fully ported because the functionality in the other schemas is overlooked in the port. An export dump file will import with errors. The foreign key constraints will fail because the referenced schema does not exist, the object in the referenced schema does not exist, or privileges from the referenced schema have not been created.

Referential integrity can be implemented, to various degrees, across databases. This can be complicated. Options are to use triggers, refreshed materialized views, or Oracle replication. Materialized views are a straightforward and balanced solution. The materialized view becomes an object in the local schema. It is refreshed using data from another database. A database link is required along with the CREATE ANY MATERIALIZED VIEW privilege. The refresh rate of the view does not make the data real-time but can be near real-time if the materialized view is refreshed frequently.

Assume that the students' database needs relatively fresh data from a government driver's license agency. The students' database needs data from a LICENSE table. In this context the license agency is the database and the LICENSE table is a master table.

The LICENSE table exists on the government agency server, AGENCY.GOV, in a database name PROD, with a username/password of SCOTT/TIGER. The LICENSE table must have a primary key. The objective is to have a local snapshot of the LICENSE table that is refreshed once every 15 minutes. This is not real-time but certainly satisfies the needs of the student's database. The remote connectivity is illustrated in Figure 3-3.

Figure 3-3. Materialized View.



The process starts by creating a Materialized View Log of the LICENSE table on the PROD.AGENCY.GOV database. This log is used by Oracle to track changes to the master table. This data in log format is used to refresh the materialized view in the student's database.

On the remote database:

```
CREATE MATERIALIZED VIEW LOG ON LICENSE;
```

On the students' database, create a link to the remote PROD database; then create the materialized view. This view will be refreshed once every 15 minutes. If the network is down, the student's application still has access to the most recent refresh.

```
CREATE DATABASE LINK prod.agency.gov
  CONNECT TO scott IDENTIFIED BY tiger
  USING 'prod.agency.gov';

CREATE MATERIALIZED VIEW STUDENT_LICENSE_RECORDS
 REFRESH FAST NEXT SYSDATE + 1/96
 AS SELECT * FROM licenses@prod.agency.gov
```

The students' tables can now reference a local snapshot of data from the government database table with license information.

Creating materialized views requires non-default privileges. The materialized view should be created with storage clauses based on the size of the snapshot.

3.3.6 Multiple Parents and DDL Migration

A child table frequently has multiple lookup tables. The use of lookup tables greatly enhances the overall integrity of the data. The STUDENTS table began with a two-character column STATE. Integrity was added to that column by creating a lookup table, STATE_LOOKUP. Once the STATE_LOOKUP table is populated, the foreign key constraint can be created. The domain of STATE abbreviations and state descriptions can grow without impact to child tables. Lookup tables are often added throughout the development process as a means to improve the integrity of the data.

From an end user's perspective, maintenance of lookup tables usually addresses the subject of end user roles and privileges. For any application there are specific roles and privileges to run the application. However, changes to tables such as STATE_LOOKUP would, and should, require special access.

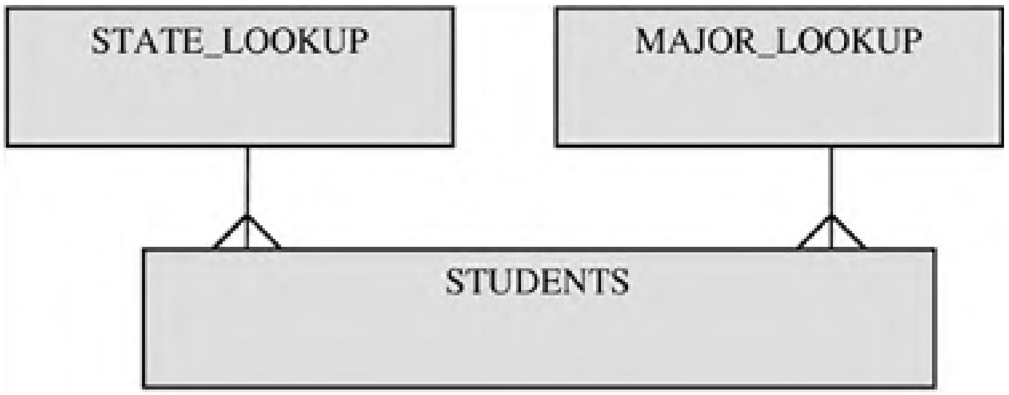
Developing the application software to support end user maintenance of lookup tables is mostly cut-and-paste. Most lookup tables have two columns. Once the first lookup maintenance screen is developed, maintenance for other lookup tables is somewhat repetitious.

Application code to display lookup data follows a standard practice of never showing the primary key column. To display student data on a screen requires joining the STUDENTS and STATE_LOOKUP tables. For each student, the student name and state description are shown but not the two character state field. HTML form elements such as drop down lists are populated with the state description, but by using the state as the option value.

```
<OPTION VALUE=state>state_description</OPTION>
```

An observation of the data outlined in the tables on p. 108 might lead one to believe that student college majors should be in a lookup table as well. This additional lookup table would restrict the domain of college major descriptions to the values controlled through a lookup table. The MAJOR_LOOKUP table would store the descriptions like "Biology" and "Math." This addition would be modeled with the entity diagram shown in Figure 3-4.

Figure 3-4. Multiple Parents.



The addition of a MAJOR_LOOKUP table to a STUDENTS and STATE_LOOKUP, illustrated in Figure 3-2, is accomplished with the following. Assume there is data present in the STATE_LOOKUP and STUDENTS table as outlined in on p. 108.

The lookup table, MAJOR_LOOKUP, must be created and populated. The following includes the CREATE script, primary key, and data load.

```
CREATE TABLE major_lookup
 (major      VARCHAR2(2)  NOT NULL,
  major_desc VARCHAR2(15) NOT NULL)
TABLESPACE student_data;

INSERT INTO major_lookup values ('UD','Undeclared');
INSERT INTO major_lookup values ('BI','Biology');
INSERT INTO major_lookup values ('MS','Math/Science');
INSERT INTO major_lookup values ('HI','History');
INSERT INTO major_lookup values ('EN','English');

ALTER TABLE major_lookup
  ADD CONSTRAINT pk_major_lookup PRIMARY KEY (major)
  USING INDEX TABLESPACE student_index;
```

The STUDENTS table must be changed. It stores the college major as VARCHAR(15). This must be replaced with a two-character field that will be a foreign key to the MAJOR_LOOKUP table. This approach creates a temporary copy (table TEMP) of the STUDENTS table, including the data. The STUDENTS table is dropped; a new STUDENTS table is created and the saved data is migrated back into the new STUDENTS table.

```
CREATE TABLE TEMP AS SELECT * FROM STUDENTS;

DROP TABLE STUDENTS;

CREATE TABLE students
 (student_id      VARCHAR2(10) NOT NULL,
  student_name    VARCHAR2(30) NOT NULL,
  college_major   VARCHAR2(2)  NOT NULL,
  status          VARCHAR2(15) NOT NULL,
  state           VARCHAR2(2),
  license_no      VARCHAR2(30))
TABLESPACE student_data;

INSERT INTO STUDENTS
SELECT student_id,
       student_name,
       decode
       (  college_major,
         'Undeclared'    , 'UD',
         'Biology'       , 'BI',
         'Math/Science' , 'MS',
         'History'       , 'HI',
         'English'       , 'EN'
       ),
       status,
       state,
       license_no
FROM temp;
```

The new STUDENTS table is populated, but with BI for Biology. The constraints are added to the STUDENTS table. This includes:

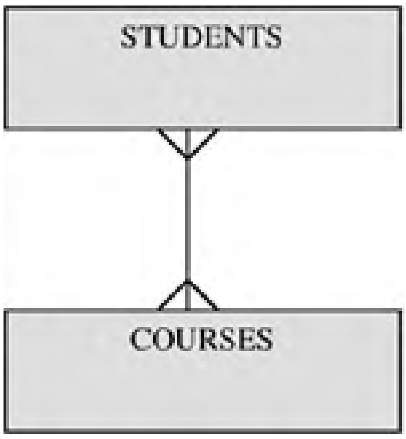
1. PRIMARY KEY
2. UNIQUE constraint on STATE, LICENSE No
3. CHECK constraint on STATE and License No
4. Foreign Key to STATE_LOOKUP
5. Foreign Key to MAJOR_LOOKUP

```
1. ALTER TABLE students
   ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
   USING INDEX TABLESPACE student_index;

2. ALTER TABLE students
   ADD CONSTRAINT uk_students_license
   UNIQUE (state, license_no)
   USING INDEX TABLESPACE student_index;

3. ALTER TABLE students
   ADD CONSTRAINT ck_students_st_lic
   CHECK ((state IS NULL AND license_no IS NULL) OR
          (state IS NOT NULL AND license_no is NOT NULL));

4. ALTER TABLE students
   ADD CONSTRAINT fk_students_state
   FOREIGN KEY (state) REFERENCES state_lookup;

5. ALTER TABLE students
   ADD CONSTRAINT fk_students_college_major
   FOREIGN KEY (college_major) REFERENCES major_lookup;
```

3.3.7 Many-to-Many Relationships

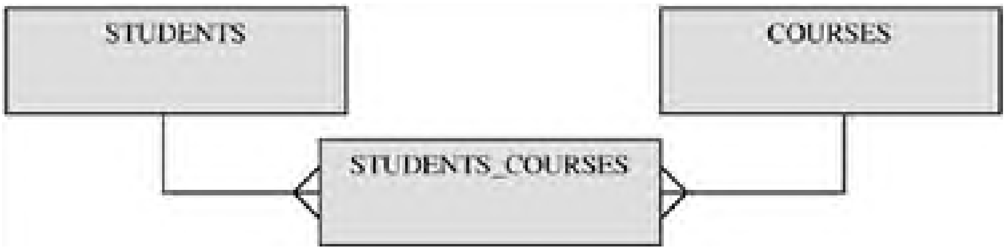
Data modeling tools will draw, at the logical level, a many-to-many relationship with the notation show in Figure 3-5.

Figure 3-5. Many-to-Many Relationship.



The model in Figure 3-5 demonstrates that a student can take several courses, while each course is taught to more than one student. There is a many-to-many relationship between students and courses. Physically, this is not implemented directly; rather, we include a cross-reference table. That cross-reference table, STUDENTS_COURSES, will contain all the courses a student takes, plus all students that take a particular course. The physical model becomes the graph in Figure 3-6.

Figure 3-6. Physical Many-to-Many Relationship.



The following types of constraints are commonly applied with the physical implementation of a many-to-many relationship.

- We have a primary key in STUDENTS.
- We have a primary key in COURSES.
- We have a CONCATENATED PRIMARY KEY in the cross-reference table, STUDENTS_COURSES.
- Part of this primary key is a foreign key to the STUDENTS table; part of it is a foreign key to the COURSES table.
- We have a foreign key in STUDENTS_COURSES that ensures each student in that table is also a student in STUDENTS. The same column in the foreign key is part of the primary key.
- We have a foreign key in STUDENTS_COURSES that ensures each course in that table is also a course in COURSES. The same column in this foreign key is part of the primary key.

When we apply these rules in the form of constraints, we will have three CREATE TABLE statements, three PRIMARY KEYS, and two FOREIGN KEY constraints.

The table description for the COURSES table is shown here:

```
Name                          Null?    Type
----------------------------- -------- ------------
COURSE_NAME                            VARCHAR2(10)
COURSE_DESC                            VARCHAR2(20)
NO_OF_CREDITS                          NUMBER(2,1);
```

The table description for the STUDENTS_COURSES table is shown here:

```
Name                          Null?    Type
----------------------------- -------- ------------
STUDENT_ID                             VARCHAR2(10)
COURSE_NAME                            VARCHAR2(10)
```

The columns of the cross-reference table contain columns that must reference the parents, in this case, STUDENTS and COURSES. This is first step. Additional columns can be added to this table, such as the professor who teaches the class and when it is taught. A cross-reference table can be just the joining columns. It can also contain additional attributes.

The DDL for this many-to-many relationship is show in Script 3-3.

Script 3-3 Many-to-Many Relationship.

```
CREATE TABLE students
 (student_id    VARCHAR2(10) NOT NULL,
  student_name  VARCHAR2(30) NOT NULL,
  college_major VARCHAR2(15) NOT NULL,
  status        VARCHAR2(20) NOT NULL,
  state         VARCHAR2(2),
  license_no    VARCHAR2(30))
TABLESPACE student_data;

CREATE TABLE courses
 (course_name   VARCHAR2(10) NOT NULL,
  course_desc   VARCHAR2(20) NOT NULL,
  no_of_credits NUMBER(2,1)  NOT NULL)
TABLESPACE student_data;

CREATE TABLE students_courses
 (student_id    VARCHAR2(10) NOT NULL,
  course_name   VARCHAR2(10) NOT NULL)
TABLESPACE student_data;

ALTER TABLE students
  ADD CONSTRAINT pk_students
  PRIMARY KEY (student_id)
  USING INDEX TABLESPACE student_index;

ALTER TABLE courses
  ADD CONSTRAINT pk_courses
  PRIMARY KEY (course_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students_courses
  ADD CONSTRAINT pk_students_courses
  PRIMARY KEY (student_id, course_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students_courses
  ADD CONSTRAINT fk_students_courses_st_id
  FOREIGN KEY (student_id)
  REFERENCES students (student_id);

ALTER TABLE students_courses
  ADD CONSTRAINT fk_students_courses_course
  FOREIGN KEY (course_name)
  REFERENCES courses (course_name);
```

3.3.8 Self-Referential Integrity

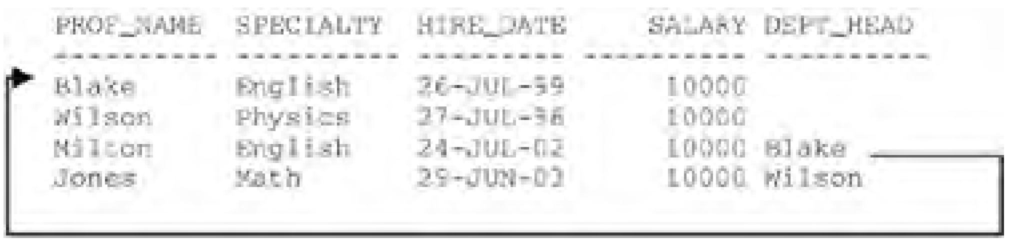
Self-referential integrity is common in many applications. Self-referential integrity allows parent-child relationships to exist between instances of the same entity. For example, all professors are in the following PROFESSORS table, including the individual department heads.

Some professors have a department head. These department heads are also professors. The following creates a table and a primary key, and then establishes a self-referential integrity constraint.

```
CREATE TABLE professors
 (prof_name     VARCHAR2(10) NOT NULL,
  specialty     VARCHAR2(20) NOT NULL,
  hire_date     DATE         NOT NULL,
  salary        NUMBER(5)    NOT NULL,
  dept_head     VARCHAR2(10))
TABLESPACE student_data;

ALTER TABLE professors
  ADD CONSTRAINT pk_professors
  PRIMARY KEY (prof_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE professors
  ADD CONSTRAINT fk_professors_prof_name
  FOREIGN KEY (dept_head)
  REFERENCES professors (prof_name);
```

This permits a scenario where a professor MAY have a department head (DEPT_HEAD) and if they have a department head, that column value MUST be an existing PROF_NAME. Figure 3-7 shows, for example, that Blake is Milton's department head.

Figure 3-7. Self-Referential Integrity Data.



```
PROF_NAME  SPECIALTY  HIRE_DATE     SALARY DEPT_HEAD
---------- ---------- --------- ---------- ----------
Blake      English    26-JUL-59      10000
Wilson     Physics    27-JUL-36      10000
Milton     English    24-JUL-02      10000 Blake
Jones      Math       29-JUN-03      10000 Wilson
```

Self-Referential Integrity, for the aforementioned data, is one level deep. The foreign key definition permits unlimited nesting. Multilevel nesting is illustrated with the following example. We create a sample table TEMP with three columns: WORKER, SALARY, and MANAGER. A worker may or may not have a manager. A manager must first be inserted as a worker. Workers can be managers of other workers who may manage other workers. The relationships among these workers and their salaries is shown as well.

```
CREATE TABLE TEMP
   (worker   VARCHAR2(10) PRIMARY KEY,
    salary   NUMBER(3),
    manager  VARCHAR2(10) REFERENCES TEMP (worker));
```

Allen manages Bill and Beth. Beth manages Cindy and Carl. Carl manages Dean and Dave.

```
Allen (salary=10) manages: Bill (salary=10), Beth (salary=10)
Beth  (salary=10) manages: Cindy (salary=5), Carl (salary=5)
Carl  (salary=5)  manages: Dean  (salary=5), Dave (salary=5)
```

This data contains multilevel relationships forming a logical tree organization. A SELECT statement using a CONNECT BY and START AT clause enables a query to return, for example, the sum of all salaries starting at a specific point in the tree. The inserts for this data are:

```
INSERT INTO TEMP values ('Allen',  10,  null  );
INSERT INTO TEMP values ('Bill' ,  10, 'Allen');
INSERT INTO TEMP values ('Beth' ,  10, 'Allen');
INSERT INTO TEMP values ('Cindy',  5,  'Beth' );
INSERT INTO TEMP values ('Carl' ,  5,  'Beth' );
INSERT INTO TEMP values ('Dean' ,  5,  'Carl' );
INSERT INTO TEMP values ('Dave' ,  5,  'Carl' );
```

The following is a START AT SELECT statement to return Beth's salary including all those who work for Beth. This is the sum of salaries for Beth, Cindy, and Carl, plus Dean and Dave who are managed by Carl: the sum is $30.00.

```
SQL> SELECT sum(salary)
  2  FROM temp
  3  START WITH worker='Beth'
  4  CONNECT BY PRIOR worker=manager;

SUM(SALARY)
-----------
         30
```

3.3.9 PL/SQL Error Handling with Parent/Child Tables

Foreign key constraint errors are captured with mapping an exception to the Oracle error, minus 2291. The following procedure inserts a student row and captures the duplicate inserts. In this case a duplicate could be a primary key constraint violation or a unique constraint error. Both generate the DUP_VAL_ON_INDEX exception.

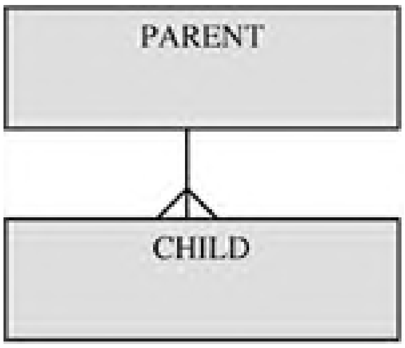
The DDL in Script 3-1 declares a CHECK constraint on STATE and LICENSE NO. The following procedure inserts a student and captures duplicate inserts that may violate the primary key or unique constraint, foreign key constraint, and check constraint.

```
CREATE OR REPLACE PROCEDURE
    insert_student(v_student_id VARCHAR2,
        v_student_name VARCHAR2,
        v_college_major VARCHAR2, v_status VARCHAR2,
        v_state VARCHAR2, v_license_no VARCHAR2)
IS
    check_constraint_violation exception;
    pragma exception_init(check_constraint_violation, -2290);

    foreign_key_violation exception;
    pragma exception_init(foreign_key_violation, -2291);

BEGIN
    INSERT INTO students VALUES
        (v_student_id, v_student_name,
         v_college_major, v_status,
         v_state, v_license_no);
    dbms_output.put_line('insert complete');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('PK or unique const violation');
    WHEN check_constraint_violation THEN
        dbms_output.put_line('check constraint violation');
    WHEN foreign_key_violation THEN
        dbms_output.put_line('foreign key violation');
END;
```

3.3.10 The Deferrable Option

In this section we use two tables that store generic parent/child data. The data model for this is shown in Figure 3-8.

Figure 3-8. Parent-Child Relationship.



The PARENT table description is:

```
Name                           Null?    Type
------------------------------ -------- ------------
PARENT_NAME                             VARCHAR2(2)
PARENT_DESC                             VARCHAR2(10)
```

The CHILD table description is:

```
Name                          Null?    Type
----------------------------- -------- ------------
CHILD_NAME                             VARCHAR2(2)
PARENT_NAME                            VARCHAR2(10)
```

The DDL, shown next, includes a DEFERRABLE option on the foreign key constraint.

```
CREATE TABLE parent
 (parent_name  VARCHAR2(2) CONSTRAINT pk_parent PRIMARY
  KEY, parent_desc  VARCHAR2(10));

CREATE TABLE child
 (child_name  VARCHAR2(10),
  parent_name VARCHAR2(2));

ALTER TABLE child ADD CONSTRAINT fk_child_parent_name
  FOREIGN KEY (parent_name)
  REFERENCES parent (parent_name) DEFERRABLE;
```

This DEFERRABLE attribute means that we can choose to defer the constraint and load a set of data into the parent and child tables, without regard to referential integrity, under the assumption that when the load completes, the data will be clean. Then a commit will automatically apply the rule on our loaded data. We can load 10 records into a child table that has no parents and then load the parents. Validation occurs on the commit. If the data we loaded violates the referential integrity rule, the transaction is rolled back.

You can do this in SQL*Plus with the following.

```
SET constraints ALL DEFERRED;
INSERT INTO child VALUES ('child_1','P1');
INSERT INTO child VALUES ('child_2','P1');
INSERT INTO child VALUES ('child_3','P2');
INSERT INTO child VALUES ('child_4','P3');
INSERT INTO child VALUES ('P1','a parent');
INSERT INTO child VALUES ('P2','a parent');
INSERT INTO child VALUES ('P3','a parent');
COMMIT;
```

You can use this functionality in a stored procedure with the following:

```
CREATE OR REPLACE PROCEDURE P IS
BEGIN
    EXECUTE IMMEDIATE 'SET constraints ALL DEFERRED';

    INSERT INTO child VALUES ('child_1','P1');
    INSERT INTO child VALUES ('child_2','P1');
    INSERT INTO child VALUES ('child_3','P2');
    INSERT INTO child VALUES ('child_4','P4');
    INSERT INTO child VALUES ('P1','a parent');
    INSERT INTO child VALUES ('P2','a parent');
    INSERT INTO child VALUES ('P3','a parent');
    COMMIT;
END P;
```

The general motivation for this is that the data comes in an inconsistent order. The aforementioned procedure inserts all child rows, then the parents. The aforementioned inserts contain hard-coded literal values. A more realistic situation could involve records read from a file using the UTL_FILE package. The input file would contain many rows out of order; that is, all child records followed by all parent records. In this case a procedure could contain a simple loop, iterate over all child inserts followed by all parent inserts, and then perform integrity checking upon a commit. In very select situations this can be a reasonable approach.

In general, there are other tools available to obviate the need for this option.

- You can write an exception handler around each child insert and should that fail, because there is no parent, you write code in the exception handler to insert a parent. Following the parent insert, you insert the child. For example, a child insert would be enclosed within an exception handler block similar to the following block, which maps the foreign key constraint violation (ORA-02291) to an exception. Upon an exception, insert a parent and then the child.

```
DECLARE
    no_parent EXCEPTION;
    PRAGMA EXCEPTION_INIT (no_parent, -2291);
    new_parent VARCHAR2 (2) := 'P6';
    new_child VARCHAR2(10) := 'child_5';
BEGIN
    INSERT INTO child VALUES (new_child,
         new_parent);
EXCEPTION
    WHEN no_parent THEN
        INSERT INTO parent VALUES (new_parent,
             'no desc');
        INSERT INTO child VALUES (new_child,
             new_parent);
END;
```

- You can store failed child records in a PL/SQL table and when you complete inserting all parents, you go back and load the child records saved in the PL/SQL table. The PL/SQL table would be derived from the child table with the following syntax:

```
TYPE temporary_table_type IS TABLE OF
    CHILD%ROWTYPE
     INDEX BY BINARY_INTEGER;
temporary_table temporary_table_type;
```

- Another option is to use a temporary table to store failed child records. This is a solution similar to using a PL/SQL Table. The PL/SQL table is manipulated like an array. The temporary table is accessed through SQL. With this solution, you load child records into this temporary repository, load all parents, then load from the temporary table into the final child table. The SQL for a temporary table is:

```
CREATE GLOBAL TEMPORARY TABLE
CHILD_TEMP
 (child_name  VARCHAR2(10),
  parent_name VARCHAR2(2))ON COMMIT DELETE ROWS;
```

The deferrable option is a powerful tool but should not be used as a standard practice because it disables the very rules that are strongly encouraged—even if the deferred state is a temporary one.

The SQL*Plus script and aforementioned stored procedure set the constraint to a deferred state with the statement:

```
SET CONSTRAINT ALL DEFERRED;
```

This SET command is one option for deferring the constraint. The other option is to replace ALL with the specific constraint name—that would be FK_CHILD_PARENT_NAME. This means your application code specifically references a constraint name. The SET CONSTRAINT ALL only affects your current transaction; the code is more generic because it does not specifically reference a constraint name. As a matter of style and preference, specific mentioning of constraint names in the code is not a recommendation.

When we create a constraint, using the following DDL, it means that we have the option to write code, or use SQL*Plus, to temporarily defer that constraint. Unless we DEFER the constraint, everything remains the same. But, if we choose to write code that temporarily breaks the rule, we must first DEFER the constraint.

```
ALTER TABLE child ADD CONSTRAINT fk_child
  FOREIGN KEY (parent_name)
  REFERENCES parent (parent_name) DEFERRABLE;
```

An option to this is to initially create the constraint in a deferred state. The DDL for this is the following.

```
ALTER TABLE child ADD CONSTRAINT fk_child
  FOREIGN KEY (parent_name)
  REFERENCES parent (parent_name) DEFERRABLE
  INITIALLY DEFERRED;
```

With this option everything is reversed. When we load data into the child and then parent tables, the constraint is deferred—this is the same as not having a constraint. Should we want to write code with the constraint enforced, as we perform each insert, then we would precede those insert statements with:

```
SET CONSTRAINT ALL IMMEDIATE;
```

3.4 Check

Declaring a database column to store a person's age starts with the following:

```
CREATE TABLE temp (age NUMBER);
```

This command will work, but the range of the data type far surpasses the domain of a person's age. The goal is to restrict one's age to the range: 1 to 125—any value outside that range is rejected. A dimension on the datatype can impose a restriction on the column so that any value, outside a three-digit number, is invalid data.

```
CREATE TABLE temp (age NUMBER(3));
```

A dimension scales down the range of valid values. Still, values far greater than 125 can be inserted—any three-digit number is possible. An age of 999 is not acceptable. In general, a CHECK constraint is used to restrict the data to a real world domain. To restrict values to integer values between 1 and 125, create a check constraint on the column.

```
CREATE TABLE temp (age NUMBER(3));

ALTER TABLE temp ADD CONSTRAINT ck_temp_age CHECK
    ((AGE>0) AND (AGE <= 125));
```

Now, the set of values we are able to insert into AGE is the set (1, 2, 125). This matches our real-world domain, with one exception, a NULL insert.

The aforementioned CREATE and ALTER TABLE statements permit a NULL. That may be within the business rules. Maybe the database does not store an age for every person. If this is the case then the aforementioned DDL is acceptable and enforces the rule.

To restrict the aforementioned AGE column to the 1–125 range and not permit nulls, attach a NOT NULL constraint. The DDL with the NOT NULL enforcement is now:

```
CREATE TABLE temp (age NUMBER(3) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_age CHECK
  ((AGE>0) AND (AGE<=125));
```

The response from a CHECK constraint violation is an ORA error with the -2290 error number.

```
SQL> insert into temp values (130);
insert into temp values (130)
*
ORA-02290: check constraint (SCOTT.CK_TEMP_AGE) violated.
```

When a row is inserted or updated and there is a check constraint, Oracle evaluates the check constraint as a Boolean expression. For the aforementioned check, the row is inserted provided the expression "the AGE is within the (1,125) range" evaluates to true. The row is rejected if it is not TRUE.

The CHECK constraint does not have to be a continuous range. Suppose we want to constrain a column value to the following boundaries.

```
(NOT NULL) AND (range in 0-10 OR 999 OR 9999)
```

The check constraint for this would be the following.

```
CREATE TABLE temp (a NUMBER);
ALTER TABLE temp ADD CONSTRAINT ck_temp_a CHECK
  (((a>=0) AND (a<=10)) OR (a=999) OR (a=9999));
```

Oracle does not evaluate the logic of the constraint defined in the DDL statement. The following constraint will never allow you to insert a row but you will certainly have no trouble creating it:

```
ALTER TABLE temp ADD CONSTRAINT ck_temp_age CHECK
  ((AGE<0) AND (AGE>=125));
```

Check constraints can be used to implement a Boolean constraint in a database column. Some databases have a Boolean type; there is no BOOLEAN table column type in Oracle—there is a BOOLEAN datatype in PL/SQL. To simulate a Boolean column use a check constraint:

```
CREATE TABLE temp(enabled NUMBER(1) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_enabled CHECK
  (enabled IN (0, 1));
```

You can use a VARCHAR2 as well. The following is another approach:

```
CREATE TABLE temp(enabled VARCHAR2(1) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_enabled CHECK
  (enabled IN ('T', 'F', 't', 'f'));
```

You can restrict a column to a discrete set of character string values. The following uses a check constraint to limit the values of a status field: 'RECEIVED,' 'APPROVED,' 'WAITING APPROVAL.'

```
CREATE TABLE temp(status VARCHAR2(16) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_status CHECK
 (status IN
 ('RECEIVED','APPROVED','WAITING APPROVAL'));
```

3.4.1 Multicolumn Constraint

A CHECK constraint can be a composite of several columns. The following table stores the dimensions of a box. We want to restrict each dimension of the box to a range within 1 and 10, plus the volume of the box must be less than 100. This means a range constraint on each column plus a constraint on the product of the dimensions.

```
CREATE TABLE box
 (length NUMBER(2) NOT NULL,
  width  NUMBER(2) NOT NULL,
  height NUMBER(2) NOT NULL);

ALTER TABLE box ADD CONSTRAINT ck_box_volume CHECK
  ((length*width*height<100) AND
   (length >  0) AND (length <= 10) AND
   (width  >  0) AND (width  <= 10) AND
   (height >  0) AND (height <= 10));
```

An insert of a zero dimension or a combination of dimensions that exceed the approved volume will generate the same constraint error. Both of the following INSERTS fail.

```
insert into box values (0,2,3);
insert into box values (8,8,8);
```

The error from each insert will be the following ORA constraint error:

```
ORA-02290: check constraint (SCOTT.CK_BOX_DIMENSION) violated.
```

You can declare multiple constraints with different names. This accomplishes the same goal of enforcing a single constraint. The only advantage is that one can see, more specifically, the exact nature of the error based on the constraint name. The following declares a separate constraint for each range constraint and one final constraint for the volume restriction.

```
CREATE TABLE box
 (length NUMBER(2) NOT NULL,
  width  NUMBER(2) NOT NULL,
  height NUMBER(2) NOT NULL);

ALTER TABLE box ADD CONSTRAINT ck_box_length CHECK
  ((length > 0) AND (length <= 10));

ALTER TABLE box ADD CONSTRAINT ck_box_width CHECK
  ((width > 0) AND (width <= 10));

ALTER TABLE box ADD CONSTRAINT ck_box_height CHECK
  ((height > 0) AND (height <= 10));

ALTER TABLE box ADD CONSTRAINT ck_box_dimension CHECK
 ((length*width*height<100));
```

The same insert statements, used earlier, still fail, but the constraint name is more specific because each constraint enforces one part of the overall rule. Repeating the inserts:

```
insert into box values (0,2,3);
insert into box values (8,8,8);
```

gives the following errors from SQL*Plus:

```
ORA-02290: check constraint (SCOTT.CK_BOX_LENGTH) violated.

ORA-02290: check constraint (SCOTT.CK_BOX_DIMENSION) violated.
```

The difference between declaring a single constraint to enforce an overall rule or separate individual constraints is a style issue.

3.4.2 Supplementing Unique Constraints

Check constraints can be used to enforce a multicolumn NOT NULL constraint that stipulates that the columns are both NULL or both NOT NULL. This is a common practice with concatenated UNIQUE constraints. A table that stores student information will have a primary key but other columns may have a unique constraint; for example, a student driver's license—this information consists of a state abbreviation and license number. A student may not have a license; in this case both columns are null. If a student has a license, then both columns are NOT NULL and are further governed by a UNIQUE constraint.

A UNIQUE constraint enforces uniqueness among all NOT NULL values. For a concatenated UNIQUE constraint, one column can be NULL; the other column may not be NULL. We may not want this condition. The rule we want to enforce is:

(both columns are NULL) OR (both columns are NOT NULL)

To stipulate the type of constraint combine a CHECK constraint with the UNIQUE constraint.

```
CREATE TABLE temp (pk NUMBER PRIMARY KEY, a NUMBER, b NUMBER);
ALTER TABLE temp
   ADD CONSTRAINT uk_temp_a_b UNIQUE (a, b);
ALTER TABLE temp ADD CONSTRAINT ck_temp_a_b
CHECK ((a IS NULL AND b IS NULL) OR
       (a IS NOT NULL AND b is NOT NULL));
```

Given the aforementioned DDL, the following inserts will violate the check constraint:

```
INSERT INTO temp VALUES (6, 1, NULL);
INSERT INTO temp VALUES (7, NULL, 1);
```

3.4.3 Students Table Example

The following DDL illustrates the motivation for a check-unique constraint combination. The STUDENTS create-table DDL below does not require license information for a student—all other columns are mandatory, but STATE and LICENSE_NO can be NULL. The STATE and LICENSE_NO, if they exist, must be unique—this is the UNIQUE constraint. The CHECK constraint enforcement is: there can never be a STATE value with a NULL LICENSE NO or a LICENSE NO value with a NULL STATE.

```
CREATE TABLE students
 (student_id      VARCHAR2(10) NOT NULL,
  student_name    VARCHAR2(30) NOT NULL,
  college_major   VARCHAR2(15) NOT NULL,
  status          VARCHAR2(20) NOT NULL,
  state           VARCHAR2(2),
  license_no      VARCHAR2(30));

ALTER TABLE students
   ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
   USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT uk_students_license
   UNIQUE (state, license_no)
   USING INDEX TABLESPACE student_index;
ALTER TABLE students ADD CONSTRAINT ck_students_st_lic
CHECK ((state IS NULL AND license_no IS NULL) OR
       (state IS NOT NULL AND license_no is NOT NULL));
```

3.4.4 Lookup Tables versus Check Constraints

Compared to using a lookup table, the following check constraint has disadvantages.

```
ALTER TABLE temp ADD CONSTRAINT ck_temp_status CHECK
 (status IN
 ('RECEIVED','APPROVED','WAITING APPROVAL'));
```

You cannot extend the list of values without dropping and recreating the constraint. You cannot write application code that will show the list of possible values, without duplicating the list within the application. The lookup table approach allows easy update and extension of the value set—that's not the case with the aforementioned check constraint.

On the other hand, to restrict person's age to a range from 1 to 125 with a lookup table is not practical. For a range-numeric rule the best choice is a check constraint.

3.4.5 Cardinality

When check constraints are used to restrict a column to a limited set of values, we may have a potential candidate for a bit map index. This only applies to those cases where the constraint is used to limit the column to a small number of possible values. Check constraints, other than numeric checks, are often columns with low cardinality. For example, consider a check constraint that restricts a column to a person's gender: 'M' or 'F' with the following:

```
CREATE TABLE temp(gender VARCHAR2(1) NOT NULL);

ALTER TABLE temp ADD CONSTRAINT ck_temp_gender CHECK
    (gender IN ('M', 'F'));
```

The column is a candidate for a bit map index if we frequently query this table based on gender. Bit map indexes are useful when the cardinality of the column is low (i.e., has only a few possible values). The following creates a bit map index on column GENDER.

```
CREATE BITMAP INDEX b_temp_gender ON TEMP
  (gender) TABLESPACE student_index;
```

3.4.6 Designing for Check Constraints

The pursuit of what columns need check constraints is sometimes overlooked. When developing new applications or migrating legacy systems to an Oracle database, the domain of an attribute often remains the same as the range of the data type. With this approach we would have permitted, in our earlier example, the insertion of someone's age to be 999 years. One reason for few check constraints in a database is simply the fact that no one knows the data well enough to state emphatically that there is some specific range of values. This is understandable.

Given this, you have two choices. Skip check constraints, or make some assumptions and the worst that can happen is that a record fails on a check constraint. In the latter case, you can always drop the constraint and recreate with the new rule, then rerun the application for which the insert failed.

Earlier we created a check constraint named CK_BOX_LENGTH. This constraint can be redefined with the following.

```
SQL> ALTER TABLE box DROP CONSTRAINT ck_box_length;

Table altered.

SQL> ALTER TABLE box ADD CONSTRAINT ck_box_length CHECK
  2     ((length > 0) AND (length <= 12));

Table altered.
```

Another reason for few check constraints appears to be the powerful enforcement capability in the client application. JavaScript in a browser is best for verifying that a field is not blank or not zero before sending the string over the network, only to have it rejected by a constraint error. It makes a lot of sense to enforce these types of rules in the client because the end user does not have to wait to realize they made an error on data entry. However, as illustrated in Figure 3-3, data goes into the database from multiple sources—not just end users and regardless of client-side rule enforcement, the rules of data integrity should still be in the database. Constraint checks will preserve data integrity for SQL*Plus operations, PL/SQL programs that may run nightly loads, and SQL*Loader runs that bring in data from other systems.

3.5 NOT NULL Constraints

The NOT NULL constraint is often referred to as a "mandatory" constraint (i.e., we say the AGE column is a mandatory value). You see this term sometimes in data modeling tools when you are asked what types of constraints you want to apply to a column.

The most common syntax for a not null constraint is to append NOT NULL within the column definition of the table. You can name a not null constraint. If you are hand-coding a lot of DDL, you may choose not to name each constraint—that can be a time-consuming task. Some tools, like Oracle Designer, automatically generate constraint names for you. This is a nice feature and should be used. The naming of constraints has been emphasized, rightfully so, within this chapter. However, naming NOT NULL constraints is just not as common as naming other constraints. In summary, it doesn't hurt to name them. The following illustrates some options.

```
CREATE TABLE temp(id NUMBER(1) NOT NULL);

CREATE TABLE temp(id NUMBER(1) CONSTRAINT nn_temp_id NOT NULL);
```

There is one significant reason why we name PRIMARY KEY, UNIQUE, FOREIGN KEY, and CHECK constraints:

> The Oracle constraint violation error message includes the name of the constraint. This is very helpful and allows us to quickly isolate the problem without having to look in the data dictionary or investigate the code that caused the error.

Oracle does not reference the constraint name in a NOT NULL constraint violation. It doesn't have to because the column name is included in the message. The following illustrates a table with three NOT NULL columns, but imagine this scenario with 20 columns. A PL/SQL code block is used to populate the table, but with one variable reset to NULL. The Oracle error message generated specifically identifies the column constraint violated. First the table:

```
CREATE TABLE temp
 (N1 NUMBER constraint nn_1 NOT NULL,
  N2 NUMBER constraint nn_2 NOT NULL,
  N3 NUMBER constraint nn_3 NOT NULL);
```

This is a short block, but it could just as well be several hundred lines of PL/SQL. The constraint error message identifies the column, saving us from isolating that part of the problem.

```
DECLARE
    v1 NUMBER := 1;
    v2 NUMBER := 2;
    v3 NUMBER := 3;
BEGIN
    v2 := null;
    INSERT INTO temp VALUES (v1, v2, v3);
END;
```

The result of running the aforementioned PL/SQL block is the following Oracle error—notice the column name, N2.

```
ORA-01400: cannot insert NULL into ("SCOTT"."TEMP"."N2")
```

So, naming the constraint served little purpose. The error says we tried to insert a NULL into the column N2. We only have to look into the PL/SQL code and trace how that variable was set to NULL.

3.6 Default Values

Consider the following table with its two columns:

```
    CREATE TABLE TEMP (id NUMBER, value NUMBER);
```

The first two INSERT statements are identical. This third inserts a NULL for the column VALUE.

```
INSERT INTO temp             VALUES (1, 100);
INSERT INTO temp (id, value) VALUES (1, 100);
INSERT INTO temp (id)        VALUES (2);
```

Is this what you want? Do you want NULL? Maybe a zero would be better. The following replaces the DEFAULT NULL with a zero.

```
CREATE TABLE TEMP (id NUMBER,
                   value VARCHAR2(10) DEFAULT 0);
```

The default for VALUE is no longer NULL. This inserts a (2, 0)

```
INSERT INTO temp (id) VALUES (2);
```

If a zero can be interpreted as no data, then use a DEFAULT as was done earlier. There are situations where a zero equates to no data. One example is a checking account. If there is a zero balance, there is no money. Other situations require a NULL. Consider an environmental biologist measuring microbes in a polluted river. On some days, the microbe count may be zero. On other days the river may be frozen. On these days an insert should be a NULL. A zero value, on days when the river is frozen, could skew any analysis of the data. For the environmentalist, there is a difference between a zero and a NULL. A NULL means no sample was taken. For the checking account, a zero is equivalent to the absence of money.

PL/SQL expressions that include NULL values can be tricky. Refer to Chapter 11, Section 11.4 for additional discussion on this topic.

3.7 Modifying Constraints

You can disable a constraint with the syntax:

```
ALTER TABLE table_name DISABLE CONSTRAINT constraint_name;
```

This leaves the constraint defined in the data dictionary. It is just not being enforced. You might do this on a development database where you need to load data that you expect will not conform to the business rules. Once the data is loaded you view it with SQL*Plus. If the data looks good you can try to enable the constraint with:

```
ALTER TABLE table_name ENABLE CONSTRAINT constraint_name;
```

If the data is bad, the ENABLE command will fail. The Oracle error message will indicate that the constraint cannot be enforced because the data does not comply with the rule. There are three options when the data is bad and you cannot enable the constraint:

- Delete the data you inserted.
- Enable the constraint using an exceptions table—this is discussed in Section 3.9, "Data Loads."
- Enable the constraint using the NOVALIDATE option. This syntax is:

  ```
  ALTER TABLE table_name ENABLE CONSTRAINT
  constraint_name NOVALIDATE;
  ```

The NOVALIDATE option enables the constraint but for future transactions. The data present in the table does not have to comply with the constraint. This can have some serious consequences. For example, many components of an application rely on business rules. A text box on an HTML form may be populated with a query that uses a primary key in the WHERE clause—this is a situation where the program assumes that one row is returned. This code will crash if it runs and multiple rows are returned.

The NOVALIDATE option poses no threat if the nature of the task is to study and analyze historical data. You can leave the old, noncompliant data in the table, enable constraints, and proceed with loading some new data that you wish to conform to your business rules. But the NOVALIDATE option on an existing production system can break many applications.

When you DISABLE a primary key or unique constraint, the index is dropped. So, if the table is large, you may see some time delay when you enable the constraint. This is the time it would take to rebuild the index. The same holds if you DROP the constraint; the index is dropped.

You use a CASCADE option when you DISABLE or DROP a primary key of unique constraint that is referenced by a foreign key. This syntax is:

```
ALTER TABLE table_name
DISABLE CONSTRAINT constraint_name CASCADE;

ALTER TABLE table_name
DROP CONSTRAINT constraint_name CASCADE;
```

You cannot accidentally corrupt the enforcement of referential integrity. If you don't realize there is a foreign key constraint and skip the cascade option, the ALTER TABLE fails with the Oracle error:

```
ORA-02272: this unique/primary key is referenced by some
foreign key
```

3.8 Exception Handling

The PL/SQL built-in exception DUP_VAL_ON_INDEX is raised whenever a SQL statement violates a primary key constraint. This is actually a PL/SQL built-in exception that is raised because you attempted to duplicate a column value for which there is a unique index—that index being the index generated on your behalf when you created the primary key. At the time such an error occurs, the Oracle error will be ORA-00001 and you can capture that error code with the PL/SQL SQLCODE built-in function. The following stored procedure implements the same functionality as the aforementioned Java procedure.

```
create or replace procedure INSERT_STUDENT
(
    v_student_name   students.student_name%TYPE,
    v_college_major  students.college_major%TYPE,
    v_status         students.status%TYPE
)
IS
BEGIN
    INSERT INTO students (student_id, student_name,
        college_major, status)
    VALUES (students_pk_seq.nextval,
            v_student_name,
            v_college_major,
            v_status);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('We have a duplicate insert');
        dbms_output.put_line('SQLERRM:'||SQLERRM);
        dbms_output.put_line('SQLCODE:'||SQLCODE);
END insert_student;
```

Should the aforementioned procedure fail due to a duplicate primary key value, the output will be the following:

```
We have a duplicate insert
SQLERRM:ORA-00001: unique constraint (SCOTT.PK_STUDENTS)
violated
SQLCODE:-1
```

The SQLCODE is an Oracle predefined function, that only has scope within an exception handler. The value of SQLCODE is not always the same as the Oracle ORA error number. In PL/SQL, your best approach to capture this specific constraint violation is to have an exception handler on the DUP_VAL_ON_INDEX exception. If your code ever enters that program scope, then you are sure you committed to either a primary key or unique constraint violation.

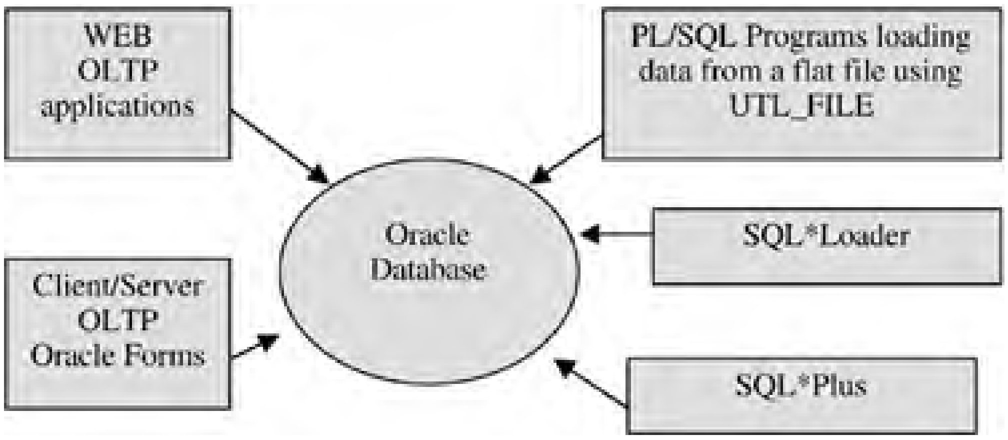
When developing applications with other languages, you need to look at the drivers. The Java code we see in Section 3.1.5, "Sequences in Code," uses the getErrorCode() method, which does not return a minus 1 but the five-digit number—for a primary key constraint violation, a 1.

We have discussed error handling from an end user perspective; that is, capture primary constraint violations, no matter how rare they might be, and respond to the user with a meaningful message, still leaving the user connected to the application.

3.9 Data Loads

Databases frequently have multiple information providers as shown in Figure 3-9.

Figure 3-9. Database Information Providers.


WEB OLTP applications
PL/SQL Programs loading data from a flat file using UTL_FILE
Oracle Database
SQL*Loader
Client/Server OLTP Oracle Forms
SQL*Plus

OLTP providers usually perform single-row inserts or updates—these transactions are usually just a few rows. Data can also be loaded in from other systems. These are batch data loads and occur when there is less OLTP activity.

Data loads typically fall into two categories. One type is a schema initialization load. This process brings data into the database for the first time and coincides with the application development. The load may be from a legacy system that is being converted to Oracle. These loads require data "scrubbing" (e.g., the legacy may require string conversions to load time/day fields into an Oracle DATE type). Constraints and indexes can be built after the data is verified and loaded.

Other batch loads occur on a periodic base. A load can initiate from a user, an operating system-scheduled job, or possibly a PL/SQL procedure scheduled through the Oracle DBMS_JOB queue.

SQL*Loader is an Oracle utility for loading fixed-format or delimited fields from an ASCII file into a database. It has a conventional and direct load option. The default option is conventional.

The direct load method disables constraints before the data load and enables them afterward. This incurs some overhead. For large amounts of data, the direct method is much faster than the conventional method. Postload processing includes not just the enabling of constraints, but the rebuilding of indexes for primary key and unique constraints.

If a direct load contains duplicates, the post process of enabling constraints and rebuilding of indexes fails. For a duplicate primary key or unique constraint, the failed state leaves the index in a "direct load" state.

Log messages in the SQL*Loader file will highlight this type of failure with the following:

```
The following index(es) on table STUDENTS were processed:
Index PK_STUDENTS was left in Direct Load State due to
ORA-0145 cannot CREATE UNIQUE INDEX; duplicate keys found
```

Following a direct load, you should check the SQL*Loader log file but also check the status of your indexes. A simple query for troubleshooting is the following:

```
SELECT index_name, table_name
FROM USER_INDEXES WHERE STATUS = 'DIRECT LOAD';
```

If you have bogus data following a direct load, you need to remove all duplicates before you can enable constraints and rebuild the indexes.

For a conventional SQL*Loader path, duplicate records are written to the SQL*Loader "bad" file with corresponding messages in the SQL*Loader "log" file. If no errors occur, then there is no "bad" file. SQL*Loader is a callable program and can be invoked in a client/server environment where the end user takes an action to load a file that is stored on the server. The mere existence of a bad file, following the load, will indicate errors during the load.

You can use SQL*Loader as a callable program to implement daily loads using a conventional path. You can use this utility to load large files with millions of rows into a database with excellent results.

Each SQL*Loader option (conventional and direct load) provides a mechanism to trap and resolve records that conflict with your primary key or any other constraint; however, direct load scenarios can be more time consuming.

Alternatives to SQL*Loader are SQL*Plus scripts and PL/SQL. You can load the data with constraints on and capture failed records through exception handling. Bad records can be written to a file using the UTL_FILE package. Bad records can also be written to a temporary table that has no constraints.

You also have the option to disable constraints, load data into a table, and then enable the constraint. If the data is bad you cannot enable the constraint. To resolve bad records, start with an EXCEPTIONS table. The exceptions table can have any name, but must have the following columns.

```
CREATE TABLE EXCEPTIONS
 (row_id     ROWID,
  owner      VARCHR2(30),
  table_name VARCHAR2(30),
  constraint VARCHAR2(30));
```

The SQL for this exceptions table is found in the ORACLE_HOME/RDBMS/ADMIN directory in the file utlecpt.sql. The RDBMS/ADMIN directory, under ORACLE_HOME, is the standard repository for many scripts including the SQL scripts to build the data dictionary catalog, scripts to compile the SYS packages, and scripts like the exceptions table.

We use the exceptions table to capture rows that violate a constraint. This capturing is done as we attempt to enable our constraint. The following TEMP table is created with a primary key.

```
CREATE TABLE TEMP
(id   VARCHAR2(5) CONSTRAINT PK_TEMP PRIMARY KEY,
 no NUMBER);
```

Insert some good data:

```
INSERT INTO temp VALUES ('AAA', 1);
INSERT INTO temp VALUES ('BBB', 2);
```

The following disables the constraint. This is done here prior to inserting new data.

```
ALTER TABLE temp DISABLE CONSTRAINT PK_TEMP;
```

Now we insert some data; in this example, this is one row that we know to be duplicate row.

```
INSERT INTO temp VALUES ('AAA', 3);
```

The following shows the error when we enable the constraint with SQL*Plus.

```
SQL> ALTER TABLE temp ENABLE CONSTRAINT pk_temp;

ORA-00001 cannot enable constraint. Unique constraint pk_temp violated.

SQL>
```

What if we had started with a million rows in TEMP and loaded another million. The task of identifying the offending rows can be tedious. Use an exceptions table when enabling constraints. The exceptions table captures the ROW ID of all offending rows.

```
ALTER TABLE temp ENABLE CONSTRAINT pk_temp EXCEPTIONS INTO exceptions;
```

The constraints are still off. All records are in TEMP, but you can identify the bad records.

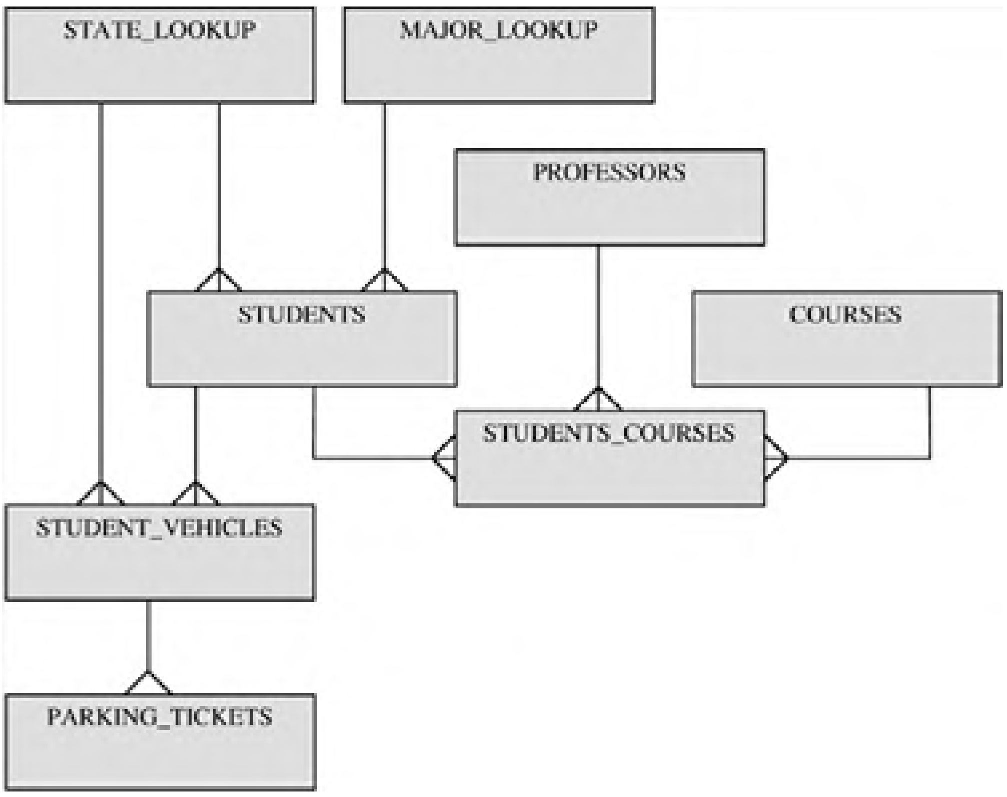
```
SELECT id, no
FROM   temp, exceptions
WHERE  exceptions.constraint='PK_TEMP'
AND    temp.rowid=exceptions.row_id;
```

This works for all types of constraint violations. You may not be able to enable constraints after a load, but you can capture the ROWID and constraint type through an exceptions table.

Chapter Four. A Data Model with Constraints

The remaining portions of this text are based on the STUDENTS data model (Figure 4-1). This model uses many of the constraints described in Chapter 3.

Figure 4-1. Students Data Model.



4.1 Entity Relationship Diagram

Figure 4-1 describes a student schema. There is a lookup table for state abbreviations and their descriptions, the STATE_LOOKUP table. Another lookup stores academic concentrations, the MAJOR_LOOKUP table.

A professor has an attribute DEPARTMENT that is enforced with a CHECK constraint rather than a lookup table. Other check constraints are on the professor's SALARY column, professor's TENURE column, and students DEGREE column.

A student can register vehicles on campus. Students register for courses from those offered in the COURSES table. The official registration list of students and classes taken by students is the STUDENTS_COURSES table.

4.2 Table Descriptions

The following includes the table descriptions for the data model.

```
SQL> desc state_lookup
 Name                            Null?    Type
 ------------------------------- -------- -----------
 STATE                           NOT NULL VARCHAR2(2)
 STATE_DESC                      NOT NULL VARCHAR2(30)

SQL> desc major_lookup
 Name                            Null?    Type
 ------------------------------- -------- -----------
 MAJOR                           NOT NULL VARCHAR2(2)
 MAJOR_DESC                      NOT NULL VARCHAR2(15)

SQL> desc students
 Name                            Null?    Type
 ------------------------------- -------- -----------
 STUDENT_ID                      NOT NULL VARCHAR2(10)
 STUDENT_NAME                    NOT NULL VARCHAR2(30)
 COLLEGE_MAJOR                   NOT NULL VARCHAR2(2)
 STATUS                          NOT NULL VARCHAR2(15)
 STATE                                    VARCHAR2(2)
 LICENSE_NO                               VARCHAR2(30)

SQL> desc student_vehicles
 Name                            Null?    Type
 ------------------------------- -------- -----------
 STATE                           NOT NULL VARCHAR2(2)
 TAG_NO                          NOT NULL VARCHAR2(10)
 VEHICLE_DESC                    NOT NULL VARCHAR2(20)
 STUDENT_ID                      NOT NULL VARCHAR2(10)
 PARKING_STICKER                 NOT NULL VARCHAR2(10)

SQL> desc parking_tickets
 Name                            Null?    Type
 ------------------------------- -------- -----------
 TICKET_NO                       NOT NULL VARCHAR2(10)
 AMOUNT                          NOT NULL NUMBER(5,2)
 STATE                           NOT NULL VARCHAR2(2)
 TAG_NO                          NOT NULL VARCHAR2(10)

SQL> desc professors
 Name                            Null?    Type
 ------------------------------- -------- -----------
 PROF_NAME                       NOT NULL VARCHAR2(10)
 SPECIALTY                       NOT NULL VARCHAR2(20)
 HIRE_DATE                       NOT NULL DATE
 SALARY                          NOT NULL NUMBER(7,2)
 TENURE                          NOT NULL VARCHAR2(3)
 DEPARTMENT                      NOT NULL VARCHAR2(10)

SQL> desc courses
 Name                            Null?    Type
 ------------------------------- -------- -----------
 COURSE_NAME                     NOT NULL VARCHAR2(10)
 COURSE_DESC                     NOT NULL VARCHAR2(20)
 NO_OF_CREDITS                   NOT NULL NUMBER(2,1)

SQL> desc students_courses
 Name                            Null?    Type
 ------------------------------- -------- -----------
 STUDENT_ID                      NOT NULL VARCHAR2(10)
 COURSE_NAME                     NOT NULL VARCHAR2(10)
 PROF_NAME                       NOT NULL VARCHAR2(10)
```

4.3 DDL

The following DDL is used to create the tables from the data model. The constraint names and types are seen in the data dictionary queries of Chapter 5.

Business rules not enforced with constraints are enforced with triggers and stored procedures in Chapters 6, 7, and 8.

```
DROP TABLE students_courses;
DROP TABLE professors;
DROP TABLE courses;
DROP TABLE parking_tickets;
DROP TABLE student_vehicles;
DROP TABLE students;
DROP TABLE state_lookup;
DROP TABLE major_lookup;

-- SEQUENCE
DROP SEQUENCE students_pk_seq;
CREATE SEQUENCE students_pk_seq START WITH 201;

--
-- STATE_LOOKUP
--
CREATE TABLE state_lookup
 (state      VARCHAR2(2)  NOT NULL,
  state_desc VARCHAR2(30) NOT NULL)
TABLESPACE student_data;

--
-- MAJOR_LOOKUP
--
CREATE TABLE major_lookup
 (major      VARCHAR2(2)  NOT NULL,
  major_desc VARCHAR2(15) NOT NULL)
TABLESPACE student_data;

--
-- STUDENTS
--
CREATE TABLE students
 (student_id      VARCHAR2(10) NOT NULL,
  student_name    VARCHAR2(30) NOT NULL,
  college_major   VARCHAR2(2)  NOT NULL,
  status          VARCHAR2(15) NOT NULL,
  state           VARCHAR2(2),
  license_no      VARCHAR2(30))
TABLESPACE student_data;

--
-- STUDENT_VEHICLES
--
CREATE TABLE student_vehicles
 (state           VARCHAR2(2)  NOT NULL,
  tag_no          VARCHAR2(10) NOT NULL,
  vehicle_desc    VARCHAR2(20) NOT NULL,
  student_id      VARCHAR2(10) NOT NULL,
  parking_sticker VARCHAR2(10) NOT NULL)
TABLESPACE student_data;

--
-- PARKING_TICKETS
--
CREATE TABLE parking_tickets
(ticket_no        VARCHAR2(10) NOT NULL,
 amount           NUMBER(5,2)  NOT NULL,
 state            VARCHAR2(2)  NOT NULL,
 tag_no           VARCHAR2(10) NOT NULL)
TABLESPACE student_data;
```

```
--
-- COURSES
--
CREATE TABLE courses
 (course_name   VARCHAR2(10) NOT NULL,
  course_desc   VARCHAR2(20) NOT NULL,
  no_of_credits NUMBER(2,1)  NOT NULL)
TABLESPACE student_data;

--
-- PROFESSORS
--
CREATE TABLE professors
 (prof_name     VARCHAR2(10) NOT NULL,
  specialty     VARCHAR2(20) NOT NULL,
  hire_date     DATE         NOT NULL,
  salary        NUMBER(7,2)  NOT NULL,
  tenure        VARCHAR2(3)  NOT NULL,
  department    VARCHAR2(10) NOT NULL)
TABLESPACE student_data;

--
-- STUDENTS_COURSES
--
CREATE TABLE students_courses
 (student_id    VARCHAR2(10) NOT NULL,
  course_name   VARCHAR2(10) NOT NULL,
  prof_name     VARCHAR2(10) NOT NULL)
TABLESPACE student_data;

----------------------------------------
-- PRIMARY KEY CONSTRAINTS
----------------------------------------
ALTER TABLE state_lookup
  ADD CONSTRAINT pk_state_lookup PRIMARY KEY (state)
  USING INDEX TABLESPACE student_index;

ALTER TABLE major_lookup
  ADD CONSTRAINT pk_major_lookup PRIMARY KEY (major)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT pk_students PRIMARY KEY (student_id)
   USING INDEX TABLESPACE student_index;

ALTER TABLE student_vehicles
  ADD CONSTRAINT pk_student_vehicles
  PRIMARY KEY (state, tag_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE parking_tickets
  ADD CONSTRAINT pk_parking_tickets
  PRIMARY KEY (ticket_no)
  USING INDEX TABLESPACE student_index;

ALTER TABLE courses
  ADD CONSTRAINT pk_courses
  PRIMARY KEY (course_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE professors
  ADD CONSTRAINT pk_professors
  PRIMARY KEY (prof_name)
  USING INDEX TABLESPACE student_index;

ALTER TABLE students_courses
  ADD CONSTRAINT pk_students_courses
  PRIMARY KEY (student_id, course_name)
  USING INDEX TABLESPACE student_index;
----------------------------------------
-- UNIQUE and CHECK CONSTRAINTS
----------------------------------------
ALTER TABLE students
   ADD CONSTRAINT uk_students_license
   UNIQUE (state, license_no)
   USING INDEX TABLESPACE student_index;

ALTER TABLE students
   ADD CONSTRAINT ck_students_st_lic
   CHECK ((state IS NULL AND license_no IS NULL) OR
          (state IS NOT NULL AND license_no is NOT NULL));

ALTER TABLE students
    ADD CONSTRAINT ck_students_status
    CHECK (status IN ('Degree','Certificate'));

ALTER TABLE professors
    ADD CONSTRAINT ck_professors_department
    CHECK (department IN ('MATH','HIST','ENGL','SCIE'));

ALTER TABLE professors
    ADD CONSTRAINT ck_professors_tenure
    CHECK (tenure IN ('YES','NO'));

ALTER TABLE professors
    ADD CONSTRAINT ck_professors_salary
    CHECK (salary < 30000);
----------------------------------------
-- FOREIGN KEY CONSTRAINTS
----------------------------------------
-- students references state_lookup
--
ALTER TABLE students
  ADD CONSTRAINT fk_students_state
  FOREIGN KEY (state) REFERENCES state_lookup;
--
-- students references major_lookup
--
ALTER TABLE students
  ADD CONSTRAINT fk_students_college_major
  FOREIGN KEY (college_major) REFERENCES major_lookup;
--
-- student_vehicles references state_lookup
--
ALTER TABLE student_vehicles
  ADD CONSTRAINT fk_student_vehicles_state
  FOREIGN KEY (state) REFERENCES state_lookup;
--
-- student_vehicles references students
--
ALTER TABLE student_vehicles
  ADD CONSTRAINT fk_student_vehicles_stud
  FOREIGN KEY (student_id) REFERENCES students;
--
-- parking_tickets references students_vehicles
--
ALTER TABLE parking_tickets
  ADD CONSTRAINT fk_parking_tickets_state_tag
  FOREIGN KEY (state, tag_no) REFERENCES student_vehicles;
--
-- students_courses references students
--
ALTER TABLE students_courses
  ADD CONSTRAINT fk_students_courses_st_id
  FOREIGN KEY (student_id)
  REFERENCES students (student_id);
--
-- students_courses references courses
--
ALTER TABLE students_courses
  ADD CONSTRAINT fk_students_courses_course
  FOREIGN KEY (course_name)
  REFERENCES courses (course_name);
--
-- students_courses references professors
--
ALTER TABLE students_courses
  ADD CONSTRAINT fk_students_courses_prof
  FOREIGN KEY (prof_name)
  REFERENCES professors (prof_name);
```

4.4 Sample Data

STATE_LOOKUP

State	State Description
CA	California
NY	New York
NC	North Carolina

MAJOR_LOOKUP

Major	Major Description
UD	Undeclared
BI	Biology
MS	Math/Science
HI	History
EN	English

STUDENTS

Student ID	Student Name	College Major	Status	State	License No
A101	John	Biology	Degree	NULL	NULL
A102	Mary	Math/Science	Degree	NULL	NULL
A103	Kathryn	History	Degree	CA	MV-32-13
A104	Steven	Biology	Degree	NY	MV-232-14
A105	William	English	Degree	NC	MV-232-15

STUDENT_VEHICLES

State	Tag No	Vehicle Desc	Student ID	Parking Sticker
CA	CD 2348	1997 Mustang	A103	C-101-AB-1
NY	MH 8709	1998 GTI	A104	C-101-AB-2
NY	JR 9837	1982 Civic	A104	C-101-AB-3

PARKING_TICKETS

Ticket No	Amount	State	Tag No
P_01	5.00	CA	CD 2348
P_02	5.00	NY	MH 8709
P_03	5.00	NY	MH 8709
P_04	5.00	NY	JR 9837

PROFESSORS

Prof Name	Specialty	Hire Date	Salary	Tenure	Department
Blake	Mathematics	07-Jul-03	1000	YES	MATH
Milton	Am History	07-Jul-02	1000	YES	HIST
Wilson	English	08-Jul-01	1000	YES	ENGL
Jones	Euro Hist	05-Aug-00	1000	YES	HIST
Crump	Ancient Hist	04-Aug-99	1000	YES	HIST

COURSES

Course Name	Course Desc	No of Credits
ENGL101	English Lit	3
MATH101	Algebra	3
HIST102	Am History	3
BIOL103	Biology	3

STUDENTS_COURSES

Student ID	Course Name	Prof Name
A101	ENGL101	Wilson
A101	MATH101	Blake
A102	HIST102	Crump
A103	HIST102	Milton
A104	BIOL103	Blake
A104	ENGL101	Wilson

Chapter Five. Viewing Constraints in the Data Dictionary

This chapter serves two purposes. First, we look at constraint information in the data dictionary, specifically the data dictionary views USER_CONSTRAINTS and USER_CONS_COLUMNS. An understanding of these views can be a basis for a broad understanding of how constraints are enforced by Oracle.

Second, this is a starting point for exploring the data dictionary. You will be able to extract a wide variety of information from the data dictionary: constraint information, triggers, stored procedures, sequence definitions. Anything you define in the database you can reverse-engineer with SQL.

The sample scripts in this chapter contain sample output assuming they were run against the DDL for the model in Chapter 4.

5.1 What You Can See

Objects that we create can always be extracted from views that begin with "USER_." If we create 10 tables in our schema, then a query of USER_TABLES will return 10 rows—each row returning attribute information about a table.

Suppose that an Oracle account user named SMITH creates a table called PAYROLL and grants all privileges on PAYROLL to the user SCOTT. Can SCOTT see PAYROLL in USER_TABLES? No, but SCOTT can see PAYROLL from ALL_TABLES. The ALL_TABLES view shows not just tables we create but also other tables to which we have been granted access.

The views we are concerned with are named with the following prefixes:

- USER
- ALL
- DBA

These prefixes are an aid in limiting the scope of what we want to see, need to see, or should be allowed to see. To see all tables we create in our schema, we query USER_TABLES; to see all tables we create plus tables to which we have received privileges from other users, we query ALL_TABLES. A user with the DBA role or SELECT_CATALOG_ROLE can query DBA_TABLES to select a list of all tables in the database. The scope defined by each prefix is the following:

USER_ Views with this prefix return a result set describing those objects created in your schema. To see a list of all the user views in this group:

[View full width]

```
SELECT view_name FROM all_views WHERE view_name
    like 'USER%';
```

ALL_ Views with this prefix return objects created in your schema plus those objects that have been created in other accounts and to which you have been granted privileges. The ALL views frequently include a column for OWNER—to reflect the owner of the object to which you have access. You won't see OWNER as a column in USER_TABLES because you are the owner of all table names in this view; however, OWNER is a column in ALL_TABLES.

DBA_ Views with this prefix provide information about the entire database. A query of OWNER and TABLE_NAME from DBA_TABLES returns the table owner and name of all tables in the database—including the base tables owned by SYS.

Not every view has a USER, ALL, and DBA prefix. There are some views that only exist in the DBA view scope. One example is the view DBA_DATA_FILES. This view is used in Chapter 3 to illustrate index space usage from a primary key index. In general, a view in USER scope has a counterpart in the ALL and DBA scope.

Access to the DBA view is sometimes given in a development environment. There is no harm allowing developers to explore the Oracle dictionary; however, you can achieve this access without releasing the DBA role. By default, with the typical CONNECT and RESOURCE roles—usually the default given to an applications developer—one cannot access these tables. You can grant the SELECT ANY TABLE privilege or SELECT_CATALOG role to a user and this will allow one to access the full data dictionary. The more one understands the data dictionary, the more in touch they are with the complex operations of the database and the more they are sensitive to such things as SQL statement tuning.

5.2 Dictionary Views: An Overview

There are some implicit relationships among the data dictionary views. Finding these relationships is often a result of experimenting with SQL queries against the data dictionary. Fortunately, there is a good deal of constancy with names used in the data dictionary.

For example, the view USER_TABLES contains a single row for each table. Each row contains detailed information about a table such as the physical storage parameters of that table. This data provides extent information telling you how the table will grow physically.

The USER_TAB_COLUMNS view contains one row for each column of each table in USER_TABLES. If a table has 10 columns, then you will find 10 rows in USER_TAB_COLUMNS, detailing information about the attributes of each column, such as the column data type. The column name TABLE_NAME is common between USER_TABLES and USER_TAB_COLUMNS so it is easy to join these views.

The information you obtain from data dictionary views is useful throughout all phases of a project. You can't possibly maintain familiarity with all data dictionary views; this is where the SQL*Plus DESCRIBE becomes most valuable. We may not be sure what columns are in a view, which makes it difficult to write a meaningful query, but we can first describe the view. This will show the column names.

If your account has the DBA role, then you can start discovering by first listing all view names with the query:

```
SELECT view_name FROM dba_views ORDER BY 1;
```

The DBA views query will return the full set of data dictionary views including the USER, ALL, and DBA views, as well as the v$ performance views (Chapter 2, Section 2.12, illustrates a sample query using the v$instance view). If you don't have the DBA role, you still have considerable access to the data dictionary. Start listing the views with:

```
SELECT view_name FROM all_views ORDER BY 1;
```

The view USER_OBJECTS is a reasonable starting point for looking into the data dictionary. There is a row in this view for everything we create. A partial description is shown next:

```
SQL> desc user_objects
 Name                           Null?    Type
 ------------------------------ -------- ---------------
 OBJECT_NAME                             VARCHAR2(128)
 OBJECT_TYPE                             VARCHAR2(18)
 CREATED                                 DATE
 STATUS                                  VARCHAR2(7)
 . . . .
```

We can select a full list of object names and their types from this view. The ALL and DBA OBJECTS views includes an OWNER column. This permits us to see who owns what. To see all objects to which you have access and who owns those objects, you can execute the following query—this is a lengthy output and includes all data dictionary objects at your disposal.

```
SELECT object_type, object_name, owner, created, status
  FROM all_objects ORDER BY 1;
```

The DDL in Chapter 4 creates objects that we can see with the following:

```
SQL> SELECT object_type, object_name
  2  FROM user_objects ORDER BY 1 DESC;

OBJECT_TYPE        OBJECT_NAME
------------------ -----------------------
TABLE              COURSES
TABLE              PARKING_TICKETS
TABLE              PROFESSORS
TABLE              STUDENTS
TABLE              STUDENT_VEHICLES
TABLE              STUDENTS_COURSES
TABLE              STATE_LOOKUP
INDEX              PK_COURSES
INDEX              UK_STUDENTS_LICENSE
INDEX          PK_PARKING_TICKETS
INDEX          PK_PROFESSORS
INDEX          PK_STATE_LOOKUP
INDEX          PK_STUDENTS
INDEX          PK_STUDENTS_COURSES
INDEX          PK_STUDENT_VEHICLES
```

Conceptually, the USER_OBJECTS view is a parent to other views that contain specific information about the objects we create. Two views of particular interest are: USER_TABLES and USER_SEQUENCES.

- A row in USER_OBJECTS, for an object of type TABLE, means there is a row in USER_TABLES. The USER_TABLES row will have more specific information about that table. The following query joins these two views. It selects the column CREATED from USER_OBJECTS plus information from USER_TABLES.

  ```
  SELECT user_objects.created,
         user_tables.table_name,
         user_tables.tablespace_name
    FROM user_objects,
         user_tables
   WHERE user_objects.object_name =
         user_tables.table_name;
  ```

  The result from this query, for the STUDENTS schema, is shown next.

  ```
  CREATED         TABLE_NAME             TABLESPACE_NAME
  --------------- ---------------------- ------------------
  18-jul-03 17:45 COURSES                STUDENT_DATA
  18-jul-03 17:45 PARKING_TICKETS        STUDENT_DATA
  18-jul-03 17:45 PROFESSORS             STUDENT_DATA
  18-jul-03 17:45 STATE_LOOKUP           STUDENT_DATA
  18-jul-03 17:45 STUDENTS               STUDENT_DATA
  18-jul-03 17:45 STUDENTS_COURSES       STUDENT_DATA
  18-jul-03 17:45 STUDENT_VEHICLES       STUDENT_DATA
  ```

- A row in USER_OBJECTS, for an object of type SEQUENCE, means there is a row in USER_SEQUENCES. The following query joins these two views where the OBJECT_TYPE = SEQUENCE.

  ```
  SELECT <columns-you-choose>
    FROM user_objects,
         user_sequences
   WHERE user_objects.object_name =
         user_sequences.sequence_name;
  ```

5.3 Constraint Views

There are two data dictionary views that provide you detailed information about constraints. These are USER_CONSTRAINTS and USER_CONS_COLUMNS.

A table can have no constraints, or many. For every constraint in a table there is a row in USER_CONSTRAINTS that describes that particular constraint, including the table name to which that constraint is applied. If you know a constraint name and you want to know the constraint type, query USER_CONSTRAINTS. This view describes the constraint definition. It does not provide you with the column name(s) on which that constraint is defined.

The USER_CONS_COLUMNS view shows the columns in a constraint. If a primary key is a concatenated key, there will be two rows for that constraint in this view. Each column in the concatenated primary key constraint will have a row, and each row will differ by POSITION—this indicates the position of a column with respect to concatenated column constraints.

The following table highlights the joining columns of these views. USER_CONSTRAINTS and USER_CONS_COLUMNS each have the column CONSTRAINT_NAME and TABLE_NAME.

USER_CONSTRAINTS	USER_CONST_COLUMNS
CONSTRAINT_NAME	**CONSTRAINT_NAME**
CONSTRAINT_TYPE	**TABLE_NAME**
TABLE_NAME	COLUMN_NAME
	POSITION

5.4 USER_CONS_COLUMNS

The USER_CONS_COLUMNS tells you what columns make up the constraint. Details about the constraint, such as whether it is a CHECK or UNIQUE constraint, are in USER_CONSTRAINTS. The SQL*Plus describe of the table is the following.

```
SQL> DESC user_cons_columns

Name                           Null?    Type
------------------------------ -------- -----------------
OWNER                          NOT NULL VARCHAR2(30)
CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
POSITION                                NUMBER
```

OWNER	This is the owner of the table in which the constraint is enforced.
CONSTRAINT_NAME	This is the name used in the DDL. You use this name to ALTER or DROP the constraint. If there is no constraint name specified in the DDL, Oracle creates the constraint with a format similar to SYS_C followed by a unique sequence of digits.
TABLE_NAME	The name of the table that has the constraint.
COLUMN_NAME	This is the column name of the constraint.
POSITION	This applies to the concatenated primary key, unique, and foreign keys. Our STUDENT DDL creates a concatenated unique constraint on columns for a student's STATE and LICENSE. For this constraint, there are two columns and two rows in this view. The row in this view, for this constraint, will have POSITION=1 for STATE and POSITION=2 for LICENSE.

5.5 USER_CONSTRAINTS

The USER_CONSTRAINTS view describes constraint types and detail attribute information on constraints. When diagnosing an Oracle constraint violation error message with a generic Oracle-generated constraint name, such as SYS_C012345, you look here to determine the type of constraint. If it is a FOREIGN KEY, then you might continue your search for the table name and column name of the parent. If it is a CHECK constraint, then you might look at the SEARCH_CONDITION to determine the rule of the CHECK constraint. When these steps are taken, you may likely look at USER_CONS_COLUMNS, using the constraint name, to determine the exact column name(s) on which the constraint is defined. The description of this view is:

```
SQL> DESC user_constraints
Name                           Null?    Type
------------------------------ -------- ----------------
OWNER                          NOT NULL VARCHAR2(30)
CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
CONSTRAINT_TYPE                         VARCHAR2(1)
TABLE_NAME                     NOT NULL VARCHAR2(30)
SEARCH_CONDITION                        LONG
R_OWNER                                 VARCHAR2(30)
R_CONSTRAINT_NAME                       VARCHAR2(30)
DELETE_RULE                             VARCHAR2(9)
STATUS                                  VARCHAR2(8)
DEFERRABLE                              VARCHAR2(14)
DEFERRED                                VARCHAR2(9)
VALIDATED                               VARCHAR2(13)
GENERATED                               VARCHAR2(14)
BAD                                     VARCHAR2(3)
RELY                                    VARCHAR2(4)
LAST_CHANGE                             DATE
```

The columns OWNER, CONSTRAINT_NAME, and TABLE_NAME have the same meaning as the view USER_CONS_COLUMNS. See the following:

CONSTRAINT_TYPE	This single character is the constraint type. C– NOT NULL or CHECK P– Primary Key U– Unique R– Foreign Key
SEARCH_CONDITION	This stores the rule of a CHECK constraint; for example, "(AGE < 125)." For a NOT NULL constraint, the value is "CHECK OR NOT NULL."
R_OWNER	A foreign key constraint must reference a column that has a primary key or unique constraint. That column can be in the table of another schema. In this case, the column identifies that schema owner. If the table is not in another schema, this value equals the current schema. This value is only set for a foreign key constraint, or when CONSTRAINT_TYPE = R.
R_CONSTRAINT_NAME	This value is only set for a foreign key constraint, or CONSTRAINT_TYPE = R. This column identifies the referenced primary key or unique constraint. A foreign key must always reference a column with a primary key or unique constraint. This value is not that column name, but rather the constraint name.
DELETE_RULE	This is only set when the constraint is a foreign key. Values for this are: CASCADE: The foreign key was created with the DELETE CASCADE rule. NO ACTION: No DELETE CASCADE was used in the foreign key definition.
STATUS	This equals ENABLED or DISABLED.
DEFERRABLE	DEFERRABLE or NOT DEFERRABLE.
DEFERRED	This is IMMEDIATE or DEFERRED. IMMEDIATE is the default. DEFERRED only applies to DEFERRABLE constraints. This is DEFERRED if the DEFERRABLE constraint is declared INITIALLY DEFERRED or if it is set to INITIALLY DEFERRED through an ALTER TABLE statement.
VALIDATED	If a constraint is enabled with the NOVALIDATE option, this column will be set to NOT VALIDATED, otherwise it is VALIDATED.

GENERATED	Set to GENERATED for Oracle-generated constraint names. Set to USER_NAME for user-generated names.
BAD	A YES indicates that the constraint specifies a century in an ambiguous manner. This can be avoided with proper use of TO_CHAR functions.
RELY	Indicates whether a constraint is enforced or unenforced.
LAST_CHANGE	The last time the constraint was enabled.
INDEX_OWNER	The owner of the index if the constraint is a primary key or unique constraint.
INDEX_NAME	The name of the index.

[Team LiB]

5.6 Data Dictionary Constraint Scripts

5.6.1 Constraints on a Table

The following query is a general script to query the constraints of a table. The use of the script accepts a table name as a command line argument.

```
SQL> @my_const_are table_name
```

Following the script text is the SQL*Plus output from the STUDENTS table.

```
-- Filename: my_const_are.sql
set verify off
column column_name format a15
column table_name format a12
set pagesize 1000
SELECT SUBSTR(A.column_name,1,30) column_name,
       DECODE(B.constraint_type,
          'P', 'PRIMARY KEY',
          'U', 'UNIQUE KEY',
          'C', 'CHECK OR NOT NULL',
          'R', 'FOREIGN KEY') constraint_type,
       A.constraint_name
FROM   user_cons_columns A,
       user_constraints B
WHERE  A.table_name = UPPER('&1')
AND    A.table_name = B.table_name
AND    A.constraint_name = B.constraint_name
AND    a.owner = b.owner
ORDER BY 2 DESC;
clear columns
```

The script file name is MY_CONST_ARE.SQL.

```
SQL> @my_const_are students

COLUMN_NAME     CONSTRAINT_TYPE   CONSTRAINT_NAME
--------------- ----------------- ----------------------
STATE           UNIQUE KEY        UK_STUDENTS_LICENSE
LICENSE_NO      UNIQUE KEY        UK_STUDENTS_LICENSE
STUDENT_ID      PRIMARY KEY       PK_STUDENTS
STATE           FOREIGN KEY       FK_STUDENTS_STATE
STUDENT_ID      CHECK OR NOT NULL SYS_C002073
STUDENT_NAME    CHECK OR NOT NULL SYS_C002074
COLLEGE_MAJOR   CHECK OR NOT NULL SYS_C002075
STATUS          CHECK OR NOT NULL SYS_C002076
```

All the constraint names that begin with SYS are NOT NULL constraints. As described in Chapter 3, an Oracle error message on these constraints includes the table name and column name.

5.6.2 Chasing a Constraint Name

You can investigate a particular constraint by joining the table and column name, from USER_CONS_COLUMNS view, with the definition of the constraint from the view USER_CONSTRAINTS. The following query is used to determine this information. If you were given a constraint with a SYS_C type name, this SQL would return the table name, column name, and constraint definition. Following the listing is the SQL*Plus session to investigate two constraints: the professor check constraint and a SYS named constraint.

```
-- Filename: i_am.sql
set verify off
column column_name format a15
column table_name format a12
column constraint_name format a20
```

```
SELECT user_constraints.constraint_name,
       user_cons_columns.table_name,
       SUBSTR(user_cons_columns.column_name,1,30)
       column_name,
       DECODE(user_constraints.constraint_type,
          'P', 'PRIMARY KEY',
          'U', 'UNIQUE KEY',
          'C', 'CHECK OR NOT NULL',
          'R', 'FOREIGN KEY') constraint_type
FROM   user_cons_columns, user_constraints
WHERE  user_constraints.constraint_name = upper('&1')
and    user_constraints.owner=user_cons_columns.owner
and    user_constraints.constraint_name=
       user_cons_columns.constraint_name;
clear columns
```

The following shows the definition of a check constraint. The file name is I_AM.SQL.

```
SQL> @i_am ck_professors_tenure

CONSTRAINT_NAME      TABLE_NAME COLUMN_NAME CONSTRAINT_TYPE
-------------------- ---------- ----------- -----------------
CK_PROFESSORS_TENURE PROFESSORS TENURE      CHECK OR NOT NULL
```

The next example tells us this is a CHECK or NOT NULL constraint on the STUDENTS STATUS column.

```
SQL> @i_am SYS_C002633

CONSTRAINT_NAME    TABLE_NAME  COLUMN_NAME CONSTRAINT_TYPE
------------------ ----------- ----------- -----------------
SYS_C002633        STUDENTS    STATUS      CHECK OR NOT NULL
```

5.6.3 CHECK Constraint Rule

A NOT NULL constraint is a type of CHECK constraint. Sometimes we need to make this determination. The NOT NULL constraint is a standard definition, but if the rule is a complicated CHECK enforcement, we need to investigate. The rule for a check constraint is in the SEARCH_CONDITION column of the USER_CONSTRAINTS view. The next query returns the rule for a check constraint.

```
-- Filename: my_rule_is.sql
set verify off
set arraysize 1
set long 75
SELECT search_condition
FROM   user_constraints
WHERE  constraint_name = upper('&1');
```

The following returns the rule for the check constraint: CK_PROFESSORS_TENURE

```
SQL> @my_rule_is ck_professors_tenure

SEARCH_CONDITION
-----------------------------------------
tenure IN ('YES','NO')
```

You can select just your check constraints by filtering on the GENERATED column. A constraint name generated by Oracle has this column set to "GENERATED NAME." Constraints you name have this column set to "USER NAME."

The following selects the table, constraint name, and check constraint rule.

```
set pagesize 0
SELECT 'T='||TABLE_NAME||' '||
       'C='||CONSTRAINT_NAME,
       SEARCH_CONDITION
FROM   USER_CONSTRAINTS
WHERE  GENERATED='USER NAME'
AND    CONSTRAINT_TYPE='C';
```

The results from this are:

```
T=PROFESSORS C=CK_PROFESSORS_SALARY
salary < 30000

T=PROFESSORS C=CK_PROFESSORS_DEPARTMENT
department IN ('MATH','HIST','ENGL','SCIE')

T=PROFESSORS C=CK_PROFESSORS_TENURE
tenure IN ('YES','NO')

T=STUDENTS C=CK_STUDENTS_ST_LIC
(state IS NULL AND license_no IS NULL) OR
        (state IS NOT NULL AND

T=STUDENTS C=CK_STUDENTS_STATUS
status IN ('Degree','Certificate')
```

5.6.4 Querying Parent Tables

The following SQL does a self-join on USER_CONSTRANTS to determine parent information.

To illustrate, the COURSES table has a foreign key called:

```
FK_COURSES_ST_ID
```

This constraint will appear in the column CONSTRAINT_NAME. This same row will have the following column values:

R_TABLE_NAME	The table name of the parent.
R_CONSTRAINT_NAME	The name of the primary key or unique constraint that the foreign key references.

```
-- Filename: my_parents_are.sql
set verify off
column foreign_key format a30 heading 'foreign key'
column parent_table format a12 heading 'parent|table'
column parent_key format a17 heading 'parent key'
SELECT a.constraint_name||
      SUBSTR(DECODE(a.delete_rule,'NO ACTION','(N)',
      'CASCADE','(C)'),1,3) foreign_key,
      b.table_name parent_table,
      a.r_constraint_name parent_key
FROM   user_constraints a, user_constraints b
WHERE  a.table_name = upper('&1')
AND    a.r_constraint_name = b.constraint_name;
clear columns

SQL> @my_parents_are students_courses

                              parent
foreign key                   table        parent key
------------------------------ ---------------------------
FK_STUDENTS_COURSES_ST_ID(N)   STUDENTS     PK_STUDENTS
FK_STUDENTS_COURSES_COURSE(N)  COURSES      PK_COURSES
FK_STUDENTS_COURSES_PROF(N)    PROFESSORS   PK_PROFESSORS
```

How would you determine the table column name that is referenced by a foreign key constraint? You know the name of the foreign key constraint. What is the parent column name?

- Query USER_CONSTRAINTS using the foreign key constraint name.
- Note the values of R_TABLE_NAME and R_CONSTRAINT_NAME.
- Query USER_CONS_COLUMNS using the R_TABLE_NAME and R_CONSTRAINT_NAME.
- Note the value(s) of COLUMN_NAME and POSITION.

5.6.5 Querying Child Tables

This script is similar to the parent script from earlier. It does a self-join on USER_CONSTRAINTS. To find a child table, look for a constraint name in the CONSTRAINT_NAME column. Then look for that same name elsewhere in the column R_CONSTRAINT_NAME—at that row the value of CONSTRAINT_NAME is a foreign key.

```
-- Filename: my_children_are.sql
set verify off
column parent_key format a15
column child_table format a16
column child_key format a28
SELECT a.r_constraint_name parent_key,
       a.table_name child_table,
       a.constraint_name||
       SUBSTR(DECODE(a.delete_rule,'NO ACTION','(N)',
       'CASCADE','(C)'),1,3) child_key
FROM   user_constraints A,
       user_constraints B
WHERE  B.table_name = upper('&1')
AND    a.r_constraint_name = b.constraint_name
ORDER BY a.table_name;
clear columns
```

Sample query:

```
SQL> @my_children_are state_lookup

PARENT_KEY      CHILD_TABLE      CHILD_KEY
--------------- ---------------- ---------------------------
PK_STATE_LOOKUP STUDENTS         FK_STUDENTS_STATE(N)
PK_STATE_LOOKUP STUDENT_VEHICLES FK_STUDENT_VEHICLES_STAT(N)
```

5.6.6 Constraint Status

You can check the ENABLE/DISABLE status of a constraint. The following disables the primary key in the STUDENTS table, plus all foreign key constraints that reference that key.

```
ALTER TABLE students DISABLE CONSTRAINT pk_students CASCADE;
```

To see what constraints are disabled:

```
SELECT table_name, constraint_name, status
 FROM user_constraints WHERE status='DISABLED'
```

The result is:

```
TABLE_NAME       CONSTRAINT_NAME          STATUS
---------------- ------------------------ --------
STUDENTS         PK_STUDENTS              DISABLED
STUDENTS_COURSES FK_STUDENTS_COURSES_ST_ID DISABLED
STUDENT_VEHICLES FK_STUDENT_VEHICLES_STUD  DISABLED
```

The primary key to STUDENTS can be enabled with the following:

```
SQL> ALTER TABLE students ENABLE CONSTRAINT pk_students;

Table altered.
```

The foreign key constraints are still disabled. They have to be enabled individually.

```
SELECT table_name, constraint_name, status
 FROM user_constraints WHERE status='DISABLED'
```

The result is:

```
TABLE_NAME      CONSTRAINT_NAME           STATUS
---------------- ------------------------ --------
STUDENTS_COURSES FK_STUDENTS_COURSES_ST_ID DISABLED
STUDENT_VEHICLES FK_STUDENT_VEHICLES_STUD  DISABLED
```

These constraints can be enabled with:

```
ALTER TABLE students_courses ENABLE CONSTRAINT
 fk_students_courses_st_id;

ALTER TABLE student_vehicles ENABLE CONSTRAINT
 fk_student_vehicles_stud;
```

5.6.7 Validated

We have a table called TEMP with duplicate values in the primary key column. This is causing some havoc in the system. The first task is to describe the table and check the status of the constraints. One possibility is that the constraint is DISABLED.

The example table description is:

```
SQL> desc temp
 Name                            Null?    Type
 ------------------------------- -------- --------------
 TEMP_ID                                  NUMBER(3)
 TEMP_DESC                                  VARCHAR2(20)
```

We check the status and last time change of the constraint.

```
SELECT table_name, constraint_name, status, last_change
FROM  user_constraints
WHERE table_name='TEMP'
AND   constraint_type='P';
```

The constraint is ENABLED. The LAST_CHANGE time can tell us if this constraint has recently been enabled and reenabled.

```
TABLE_NAME     CONSTRAINT_NAME    STATUS   LAST_CHANGE
-------------- ------------------ -------- ------------
TEMP           PK_TEMP            ENABLED  07-aug 21:27
```

The constraint is enabled but there are duplicates. This has to be a deferrable constraint. Data was loaded and the constraint was enabled with the NOVALIDATE option. We can verify this with the following:

```
SELECT table_name, constraint_name,
       deferred, deferrable, validated
FROM   user_constraints
WHERE deferrable='DEFERRABLE';
```

The result from this is the following:

```
TABLE_NAME CONSTRAINT DEFERRED  DEFERRABLE     VALIDATED
---------- ---------- --------- -------------- -------------
TEMP       PK_TEMP    IMMEDIATE DEFERRABLE     NOT VALIDATED
```

The problem needs correction. There are duplicates and they must be removed. We can use an EXCEPTIONS table as described next:

```
SQL> desc exceptions
 Name                            Null?    Type
 ------------------------------- -------- ------------
 ROW_ID                                   ROWID
 OWNER                                    VARCHAR2(30)
 TABLE_NAME                                  VARCHAR2(30)
 CONSTRAINT                                  VARCHAR2(30)
```

You can enable the constraint using the exceptions table. The offending rows are recorded in the EXCEPTIONS table.

You should delete any existing rows in the exceptions table first.

```
SQL> ALTER TABLE temp ENABLE CONSTRAINT pk_temp
  2  EXCEPTIONS INTO EXCEPTIONS;

ORA-02437: cannot validate (SCOTT.PK_TEMP) - primary key
  violated
```

The EXCEPTIONS table has each row that is a duplicate. First, let's query the problem table.

```
SQL> SELECT * FROM temp;

   TEMP_ID TEMP_DESC
---------- ------------------
         1 Record One
         1 Record Two
         3 Record Three
         4 Record Four
```

For this small table, the duplicate rows are easily detected. A larger table would require a join of EXCEPTIONS and TEMP:

```
SELECT temp_id,temp_desc
FROM   temp, exceptions
WHERE  temp.rowid=exceptions.row_id;
```

The result from this query is:

```
TEMP_ID TEMP_DESC
------- --------------------
      1 Record One
      1 Record Two
```

The table can be corrected with an update changing TEMP_ID to "2" for "Record Two." Once this is done all column values will be unique. The next step is to set the constraint to the state VALIDATE.

```
UPDATE temp SET temp_id=2 WHERE temp_desc='Record Two';

ALTER TABLE temp ENABLE CONSTRAINT pk_temp;
```

The constraint is enabled and validated. To rerun the aforementioned query:

```
SELECT table_name, constraint_name,
       deferred, deferrable, validated
FROM   user_constraints
WHERE deferrable='DEFERRABLE';
```

We see that the constraint is VALIDATED.

```
TABLE_NAME CONSTRAINT DEFERRED  DEFERRABLE     VALIDATED
---------- ---------- --------- -------------- -------------
TEMP       PK_TEMP    IMMEDIATE DEFERRABLE     VALIDATED
```

Chapter Six. Row Trigger Mechanics

This chapter covers the mechanics of row triggers. Most examples are given with INSERT triggers. Row triggers can fire from INSERT, UPDATE, and DELETE statements. The objective is to cover the mechanics of row triggers first, and then treat differences among insert-row, update-row, and delete-row triggers.

6.1 Introduction

A trigger is a compiled procedure stored in the database. The language you use is PL/SQL. You code and compile a trigger in the same manner you code stored procedures. The following is the SQL*Plus session that creates and demonstrates a simple Insert Row trigger. This trigger calls DBMS_OUTPUT to print "executing temp_air" for each row inserted.

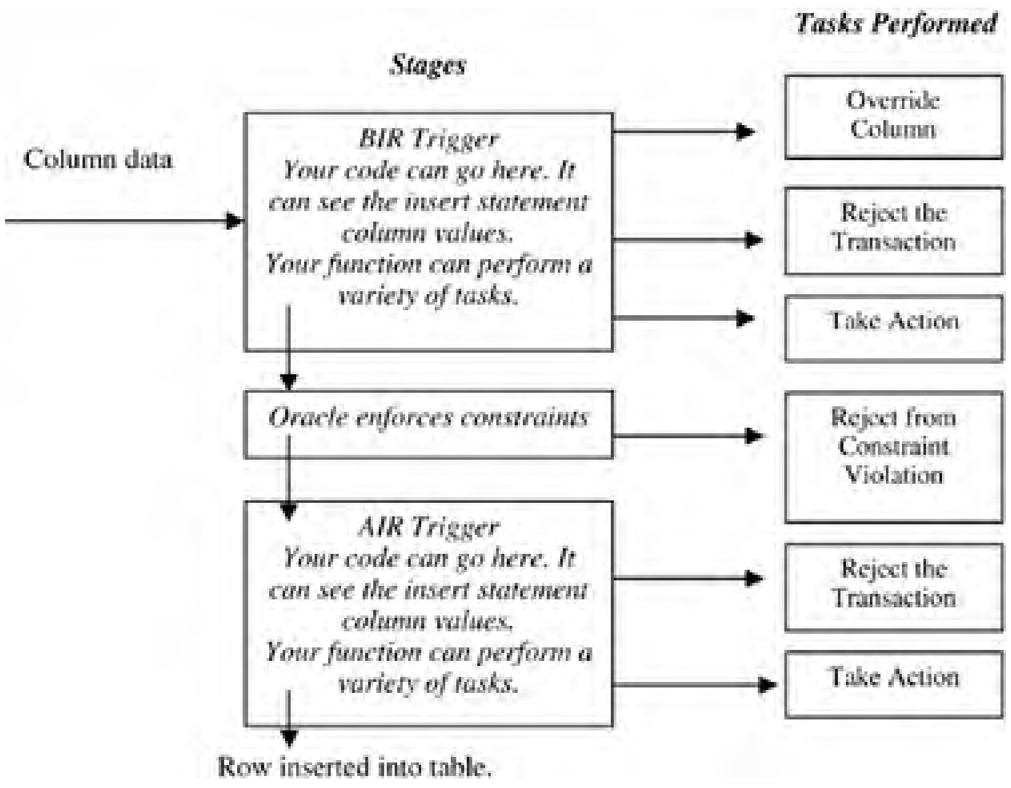
```
SQL> set feedback off
SQL> CREATE TABLE temp (N NUMBER);
SQL> CREATE OR REPLACE TRIGGER temp_air
  2  AFTER INSERT ON TEMP
  3  FOR EACH ROW
  4  BEGIN
  5  dbms_output.put_line('executing temp_air');
  6  END;
7   /
8   SQL> INSERT INTO temp VALUES (1);       -- insert 1 row
executing temp_air
SQL> INSERT INTO temp SELECT * FROM temp;  -- insert 1 row
executing temp_air
SQL> INSERT INTO temp SELECT * FROM temp;  -- inserts 2 rows
executing temp_air
executing temp_air
SQL>
```

The third INSERT statement inserted two rows into TEMP, even though this was a single SQL statement. Most insert SQL statements insert a single row; however, as shown earlier, multiple rows can be inserted with one statement.

6.2 Before versus After

The insert of a single row has three stages. Figure 6-1 illustrates each stage. There are two stages in which you can inject your PL/SQL code. The middle stage is the Oracle constraint enforcement. The first stage is the execution of the BEFORE-INSERT trigger. The trigger code at this stage runs BEFORE Oracle's constraint enforcement and is called a BEFORE-INSERT-ROW (BIR) trigger. You can write code that executes AFTER Oracle's constraint enforcement. This is the AFTER-INSERT-ROW (AIR) trigger.

Figure 6-1. Row Insert Trigger.


Tasks Performed
Stages
Column data
BIR Trigger Your code can go here. It can see the insert statement column values. Your function can perform a variety of tasks.
Override Column
Reject the Transaction
Take Action
Oracle enforces constraints
Reject from Constraint Violation
AIR Trigger Your code can go here. It can see the insert statement column values. Your function can perform a variety of tasks.
Reject the Transaction
Take Action
Row inserted into table.

From an initial look at Figure 6-1 it seems that you can choose to perform an operation that executes before constraint enforcement or after the enforcement. In general, this is true.

Depending on what you want to accomplish you may be required to put your code in a before-constraint trigger, the first stage. For example, a trigger code that "overrides column" values must execute in a BEFORE-INSERT row trigger. In some situations, you may have a choice to use a before or after row trigger, but choose the after trigger because you want your code to execute after Oracle's constraint enforcement.

Each trigger in Figure 6-1 has the capability to "see the insert statement column values." This is a valuable feature. This feature allows a row trigger, for example, to raise an error if a column value violates a business rule.

Figure 6-1 shows the possible tasks that can be performed (Tasks Performed) for the various stages of the insert statement. There are three stages: BEFORE-INSERT row trigger execution, Oracle constraint enforcement, and AFTER-INSERT row trigger execution. The action taken by Oracle's constraint enforcement is to raise an error or accept the data. The Task Performed by application developed triggers falls into three general categories:

- Override Column (Before Row Trigger only)
- Reject Transaction (Before and After Trigger)
- Take Action (Before or After Trigger)

"Override Column" refers to a trigger changing the content of a value from the INSERT statement. For example, a BIR trigger can truncate a value for a DATE column. This means that an INSERT statement that uses SYSDATE in the statement will have that value truncated when it is inserted in the database.

"Reject the Transaction," in Figure 6-1, refers to the application enforcement of a complex rule. A complex rule is any rule that is not easily or possibly enforced with an Oracle CHECK constraint and can only be enforced procedurally. A trigger rejects an insert by calling the built-in function RAISE_APPLICATION_ERROR.

"Take Action," in Figure 6-1, refers to any action the trigger may take. This includes a print with DBMS_OUTPUT for debug purposes. Row triggers can send signals to other processes using the Oracle DBMS_ALERTS package. Row triggers sometimes populate global temporary tables with before and after row values for later use by after statement level triggers. Chapter 7 continues with this topic in Section 7.4, "Processing Row Captured Data."

PREVIOUS NEXT

6.3 Insert Row Trigger Syntax

The Insert Row trigger has the following syntax.

```
CREATE OR REPLACE TRIGGER trigger_name
AFTER|BEFORE INSERT ON table_name
FOR EACH ROW
[WHEN (Boolean expression)]
DECLARE
    Local declarations
BEGIN
    Trigger Body written PL/SQL
END;
```

TRIGGER_NAME

Use trigger names that identify the table name and trigger type. A PL/SQL run time error will generate a PL/SQL error message and reference the trigger name and line number. The following Oracle error indicates that Line 5 in the AFTER-INSERT row trigger, on the STUDENTS table, has a divide-by-zero error.

```
ORA-01476: divisor is equal to zero
ORA-06512: at "SCOTT.STUDENTS_AIR", line 5
ORA-04088: error during execution of trigger
  'SCOTT.STUDENTS_AIR'
```

The counting of line numbers begins at the keyword DECLARE. If there is no DECLARE section, then the BEGIN statement is line number 1. Trigger names are found in the column TRIGGER_NAME of USER_TRIGGERS. Trigger names used in this text are derived from the table name, trigger_type, and triggering event. The syntax is:

```
trigger_name = table_name_[A|B] [I|U|D] [R|S]
```

trigger_name	Trigger names are limited to 30 characters. You have to abbreviate the table name to append trigger attribute information. Long table names should have a consistent abbreviation. The table name is used in the naming of foreign key constraints and trigger names. A consistent abbreviation shortens the troubleshooting process.
[A|B]	Indicates the BEFORE or AFTER part of the trigger type.
[I|U|D]	Indicates the triggering event. The triggering event can be "INSERT." It can be "UPDATE." It can be "INSERT OR UPDATE OR DELETE."
[R|S]	Indicates the ROW or STATEMENT part of the trigger type.

Examples of trigger names:

Table	Trigger Type	Triggering Event	Trigger Name
STUDENTS	BEFORE EACH ROW	INSERT	students_bir
STUDENTS	AFTER EACH ROW	INSERT OR UPDATE	students_aiur
STUDENTS	AFTER STATEMENT	INSERT	students_ais

AFTER|BEFORE INSERT ON TABLE_NAME

This clause tells Oracle when to execute the trigger. It can be BEFORE or AFTER Oracle's integrity constraint checks. You can designate a BEFORE or AFTER trigger to fire on multiple statement types, examples are:

```
BEFORE INSERT OR UPDATE on table_name
BEFORE INSERT OR UPDATE OR DELETE on table_name
AFTER INSERT OR DELETE on table_name
```

Case logic in the trigger body can isolate a section of code to a particular SQL statement. The Oracle package DBMS_STANDARD includes four functions that are intended for this purpose:

```
PACKAGE DBMS_STANDARD IS
    FUNCTION inserting RETURN BOOLEAN;
    FUNCTION updating RETURN BOOLEAN;
    FUNCTION updating (colnam VARCHAR2) RETURN BOOLEAN;
    FUNCTION deleting RETURN BOOLEAN;
    etc,
END DBMS_STANDARD;
```

Use of the aforementioned functions in any context other than a trigger body evaluates to NULL. You do not use the package name when referencing these functions. The following illustrates a trigger using a CASE construct and the DBMS_STANDARD functions.

```
CREATE OR REPLACE TRIGGER temp_aiur
AFTER INSERT OR UPDATE ON TEMP
FOR EACH ROW
BEGIN
    CASE
    WHEN inserting THEN
        dbms_output.put_line
            ('executing temp_aiur - insert');
    WHEN updating THEN
        dbms_output.put_line
            ('executing temp_aiur - update');
    END CASE;
END;
```

An UPDATE ROW trigger can specify the columns being updated as a condition for firing the trigger. The syntax is:

```
OF column_name [,column_name]
```

For example, the following trigger fires only when columns M or P are included in the UPDATE statement.

```
CREATE OR REPLACE TRIGGER temp_aur
AFTER INSERT OR UPDATE OF M, P ON TEMP
FOR EACH ROW
BEGIN
    dbms_output.put_line
       ('after insert or update of m, p');
END;
```

DECLARE

As with any PL/SQL block, this is not necessary if there are no local declarations.

WHEN (BOOLEAN EXPRESSION)

This clause is optional and can be used to filter the condition for when you want to fire the trigger.

Why would you use this? The WHEN option can be used on any ROW level trigger. Consider updating a table with many rows. A row level trigger can impact performance. If a million rows are updated, one million executions of the trigger will be noticed. The performance impact from the row trigger could be reason enough to remove it from the application.

However, there may be special circumstances in which the trigger is needed. The WHEN clause provides an opportunity to control the trigger execution—to have it fire on short transactions and not on massive updates.

Within the parentheses of the WHEN clause, the reference to column values is with the following syntax:

NEW.COLUMN_NAME	This is the syntax for referencing a column in the WHEN clause of an INSERT or UPDATE trigger.
OLD.COLUMN_NAME	This can be used in the WHEN clause of UPDATE and DELETE ROW triggers. This does not evaluate in INSERT ROW triggers.

To illustrate, the following is an AFTER INSERT row trigger that fires only when the column value for N is equal to 0.

```
CREATE OR REPLACE TRIGGER temp_air
AFTER INSERT ON TEMP
FOR EACH ROW
WHEN (NEW.N = 0)
BEGIN
    dbms_output.put_line('executing temp_air');
END;
```

The aforementioned trigger will fire with this next SQL statement:

```
INSERT INTO TEMP VALUES (0);
```

The aforementioned trigger will not fire with the following two statements:

```
INSERT INTO TEMP VALUES (2);
INSERT INTO TEMP VALUES (NULL);
```

The value of NEW.column is either the value included in the SQL statement, NULL, or the value from the column DEFAULT. Consider the TEMP TABLE:

```
CREATE TABLE temp (N NUMBER DEFAULT 0, M NUMBER);
```

The aforementioned trigger will fire for the following SQL statement. This is because NEW.N is equal to the DEFAULT, which is 0. The trigger fires only when NEW.N is zero.

```
INSERT INTO TEMP (M) VALUES (3);
```

You can code OLD.column in an INSERT trigger. It is not a syntax error and evaluates to NULL. The following illustrates a trigger with a triggering event of INSERT OR UPDATE. The WHEN clause stipulates that the trigger fires on either of the following conditons:

OLD.N=0 AND NEW.N=1	This expression does not evaluate on any INSERT statements because OLD.N is NULL. It may be TRUE or FALSE on UPDATE statements.
OR	
NEW.N=1	This evaluates on INSERT and UPDATE statements.

```
CREATE OR REPLACE TRIGGER temp_biur
BEFORE INSERT OR UPDATE ON TEMP
FOR EACH ROW
WHEN (OLD.N = 0 AND NEW.N=1 OR NEW.N=1)
 BEGIN
    dbms_output.put_line('executing temp_biur');
END;
```

The aforementioned trigger fires on INSERTs when N is 1 and on UPDATEs when the value of N changes from 0 to 1.

The WHEN clause can include any Boolean expression. It may include PL/SQL function calls. The following illustrates the use of the SQL function BETWEEN. This trigger fires only when the inserted value is between 1 and 10.

```
CREATE OR REPLACE TRIGGER temp_aur
AFTER UPDATE ON TEMP
FOR EACH ROW
WHEN (NEW.N BETWEEN 1 AND 10)
BEGIN
    dbms_output.put_line('executing temp_aur');
END;
```

The WHEN clause is for ROW triggers only.

6.4 Trigger Body

The trigger body is written in PL/SQL. There are PL/SQL expressions that only have meaning within a trigger body. These topics are covered in the following paragraphs.

A trigger body can refer to two states of a column value. The syntax for each state is:

```
:NEW.COLUMN_NAME
:OLD.COLUMN_NAME
```

:NEW.COLUMN_NAME is the syntax for referencing a column value within the body of the row INSERT and UPDATE triggers. This expression evaluates to NULL in DELETE row triggers. It evaluates to the value included in the SQL INSERT or UPDATE statement.

If the SQL statement does not reference a column, the following rules apply:

- For INSERT statements, :NEW.COLUMN_NAME is either NULL or the value of the DEFAULT used in creating the table.
- For UPDATE statements, :NEW.COLUMN_NAME is the value currently in the table.

:OLD.COLUMN_NAME is valid for UPDATE and DELETE row triggers only. This evaluates to NULL for INSERT triggers. The :OLD.COLUMN_NAME evaluates to the value currently in the table.

The values of :OLD.COLUMN_NAME and :NEW.COLUMN_NAME expressions are identical for BEFORE and AFTER row triggers. The choice of a BEFORE or AFTER row trigger is a choice to execute the trigger before or after Oracle constraint enforcement. Visibility to :OLD and NEW values is identical with each. One caveat is that you can change the value of :NEW.COLUMN_NAME in BEFORE row triggers, not in AFTER row triggers.

The following illustrates a table that has BEFORE UPDATE and AFTER UPDATE row triggers. It prints the old and after column values.

```
CREATE TABLE TEMP(N NUMBER, M NUMBER DEFAULT 5);
CREATE OR REPLACE TRIGGER temp_bur
BEFORE UPDATE ON TEMP
FOR EACH ROW
BEGIN
   dbms_output.put_line('BUR old N:'||:old.n|| ' M:'||:old.M);
   dbms_output.put_line('BUR new N:'||:new.n|| ' M:'||:new.M);
END;

CREATE OR REPLACE TRIGGER temp_aur
AFTER UPDATE ON TEMP
FOR EACH ROW
BEGIN
   dbms_output.put_line('AUR old N:'||:old.n|| ' M:'||:old.M);
   dbms_output.put_line('AUR new N:'||:new.n|| ' M:'||:new.M);
END;
```

The OLD and NEW values are the same for BEFORE and AFTER row triggers.

```
SQL> INSERT INTO TEMP (n) VALUES (1);
SQL> UPDATE TEMP SET n=n+1 WHERE n>=1;
BUR old: N=1 M=5
BUR new: N=2 M=5
AUR old: N=1 M=5
AUR new: N=2 M=5
SQL> UPDATE TEMP SET m=2 WHERE M=5;
BUR old: N=2 M=5
BUR new: N=2 M=2
AUR old: N=2 M=5
AUR new: N=2 M=2
```

The BEFORE UPDATE row trigger is changed. It increments the :NEW.N. This effectively changes the value inserted into the database. This trigger has one line added, the statement to increment :NEW.N.

```
CREATE OR REPLACE TRIGGER temp_bur
BEFORE UPDATE ON TEMP
FOR EACH ROW
BEGIN
    :NEW.N := :NEW.N + 1;
    dbms_output.put_line('BUR old N:'||:old.n|| ' M:'||:old.M);
    dbms_output.put_line('BUR new N:'||:new.n|| ' M:'||:new.M);
END;
```

The results of the same INSERT and UPDATE statements show that N was inserted into the database with an incremented value. The first UPDATE in the following statement increments: N to N + 1, from 1 to 2. The trigger increments N again to a value of 3. The second UPDATE statement does not reference the column N, yet it is incremented in the column.

```
SQL> INSERT INTO TEMP (n) VALUES (1);
SQL> UPDATE TEMP SET n=n+1 WHERE n>=1;
BUR old: N=1 M=5
BUR new: N=3 M=5
AUR old: N=1 M=5
AUR new: N=3 M=5
SQL> UPDATE TEMP SET m=2 WHERE M=5;
BUR old: N=3 M=5
BUR new: N=4 M=2
AUR old: N=3 M=5
AUR new: N=4 M=2
```

6.5 Example Row Triggers

Consider the following table:

```
CREATE TABLE TEMP(X NUMBER, Y NUMBER, Z NUMBER DEFAULT 5);
```

1. Write a trigger that fires ONLY under the following conditions:
   - UPDATE when Y changes from NULL to a NOT NULL value.
   - INSERT when X is between 1 and 10.

```
CREATE OR REPLACE TRIGGER temp_aiur
AFTER INSERT OR UPDATE OF Y ON TEMP
FOR EACH ROW
WHEN (OLD.Y IS NULL and NEW.Y IS NOT NULL
      OR NEW.X BETWEEN 1 AND 10)
BEGIN
    CASE
    WHEN inserting THEN
        dbms_output.put_line('X := '||:new.x);
    WHEN updating THEN
       dbms_output.put_line
          ('Y is reset from NULL');
    END CASE;
END;
```

2. Write a trigger to print the current values in a row being deleted.

```
CREATE OR REPLACE TRIGGER temp_adr
AFTER DELETE ON TEMP
FOR EACH ROW
BEGIN
    dbms_output.put_line
       (:old.x||' '||:old.y||' '||:old.z);
END;
```

3. Write all six possible row level triggers: BEFORE and AFTER ROW for INSERT, UPDATE, and DELETE.

Trigger Type	Trigger Template Code

BEFORE INSERT	CREATE OR REPLACE TRIGGER temp_bir
	BEFORE INSERT ON TEMP
	FOR EACH ROW
	BEGIN
	    dbms_output.put_line('executing temp_bir');
	END;

AFTER INSERT	CREATE OR REPLACE TRIGGER temp_air
	AFTER INSERT ON TEMP
	FOR EACH ROW
	BEGIN
	    dbms_output.put_line('executing temp_air');
	END;

BEFORE UPDATE (see note)	CREATE OR REPLACE TRIGGER temp_bur
	BEFORE UPDATE ON TEMP
	FOR EACH ROW
	BEGIN
	dbms_output.put_line('executing temp_bur');
	END;

AFTER UPDATE (see note)	CREATE OR REPLACE TRIGGER temp_aur
	AFTER UPDATE ON TEMP
	FOR EACH ROW
	BEGIN
	    dbms_output.put_line('executing temp_aur');
	END;

BEFORE DELETE	CREATE OR REPLACE TRIGGER temp_bdr
	BEFORE DELETE ON TEMP
	FOR EACH ROW
	BEGIN
	    dbms_output.put_line('executing temp_bdr');
	END;

AFTER DELETE	CREATE OR REPLACE TRIGGER temp_adr
	AFTER DELETE ON TEMP
	FOR EACH ROW
	BEGIN
	    dbms_output.put_line('executing temp_adr');
	END;

Note. The OF COLUMN_NAME clause is optional on UPDATE row and UPDATE statement level triggers (statement level triggers are covered in Chapter 7). The WHEN (Boolean expression) is optional with all ROW triggers.

4. Write two triggers that accomplish the same as the prior six triggers.

```
CREATE OR REPLACE TRIGGER temp_biudr
BEFORE INSERT OR UPDATE OR DELETE ON TEMP
FOR EACH ROW
BEGIN
    CASE
    WHEN inserting THEN
        dbms_output.put_line('inserting before');
    WHEN updating THEN
        dbms_output.put_line('updating before');
    WHEN deleting THEN
        dbms_output.put_line('deleting before');
    END CASE;
END;

CREATE OR REPLACE TRIGGER temp_aiudr
AFTER INSERT OR UPDATE OR DELETE ON TEMP
FOR EACH ROW
BEGIN
    CASE
    WHEN inserting THEN
        dbms_output.put_line('inserting after');
    WHEN updating THEN
        dbms_output.put_line('updating after');
    WHEN deleting THEN
        dbms_output.put_line('deleting after');
    END CASE;
END;
```

6.6 A Table with Oracle Constraints and Business Rules

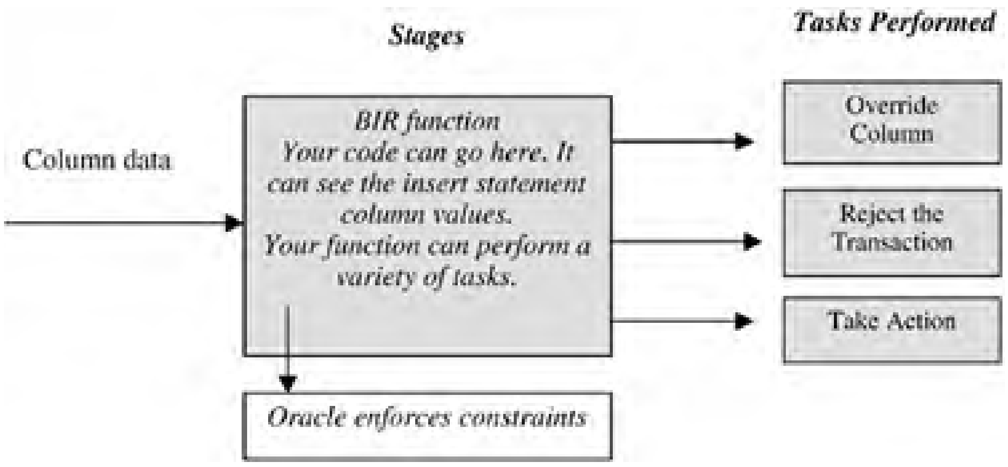
In this section a set of business rules is applied to INSERTS on the PROFESSORS table. We will initially enforce the rules with a BEFORE INSERT row trigger. This trigger will perform a variety of tasks. It will:

- Override column values upon insert.
- Enforce a strict business rule that may potentially result in the transaction being rejected through a call to RAISE_APPLICATION_ERROR.
- Take an action. In this case print using DBMS_OUTPUT.

6.6.1 The Environment

The location of this logic is illustrated in Figure 6-2. The grayed area indicates where this trigger code fits into the overall INSERT statement process.

Figure 6-2. Location of Trigger Code.


Stages
Tasks Performed
Column data
BIR function
Your code can go here. It can see the insert statement column values.
Your function can perform a variety of tasks.
Override Column
Reject the Transaction
Take Action
Oracle enforces constraints

The business rules in this example are being applied to the PROFESSORS table. Here is a description of that table:

```
SQL> desc professors
 Name                          Null?    Type
 ----------------------------- -------- ------------
 PROF_NAME                     NOT NULL VARCHAR2(10)
 SPECIALTY                     NOT NULL VARCHAR2(20)
 HIRE_DATE                     NOT NULL DATE
 SALARY                        NOT NULL NUMBER(7,2)
 TENURE                        NOT NULL VARCHAR2(3)
 DEPARTMENT                    NOT NULL VARCHAR2(10)
```

and the CHECK constraints for this table are:

- The name of the department is restricted to a limited set.

```
ALTER TABLE professors
    ADD CONSTRAINT ck_professors_department
    CHECK (department IN
('MATH','HIST','ENGL','SCIE'));
```

- The tenure field is YES or NO.

```
ALTER TABLE professors
    ADD CONSTRAINT ck_professors_tenure
    CHECK (tenure IN ('YES','NO'));
```

- The salary has an upper limit of 30,000.00.

```
ALTER TABLE professors
    ADD CONSTRAINT ck_professors_salary
    CHECK (salary < 30000);
```

6.6.2 The Procedural Constraints to Enforce

The following lists the rules we are asked to enforce. Rules are grouped using one the categories from in Figure 6-1 and 6-2.

OVERRIDE COLUMN RULES

- *Truncate the HIRE_DATE to ensure that all inserted dates have a zero for hours, minutes, and seconds.*
- *Convert the DEPARTMENT to uppercase. This is just to ensure that an INSERT of 'Math' is converted to 'MATH.'*
- *Round all new salaries to the nearest dollar.*

REJECT THE TRANSACTION RULE

- *Reject any INSERT where the salary of the professor exceeds a $10,000 limit and the department is 'ENGL.' Note: All other salaries are constrained to the CHECK constraint that has a limit of $30,000.*

TAKE ACTION RULE

- *Write a message using DBMS_OUTPUT.*

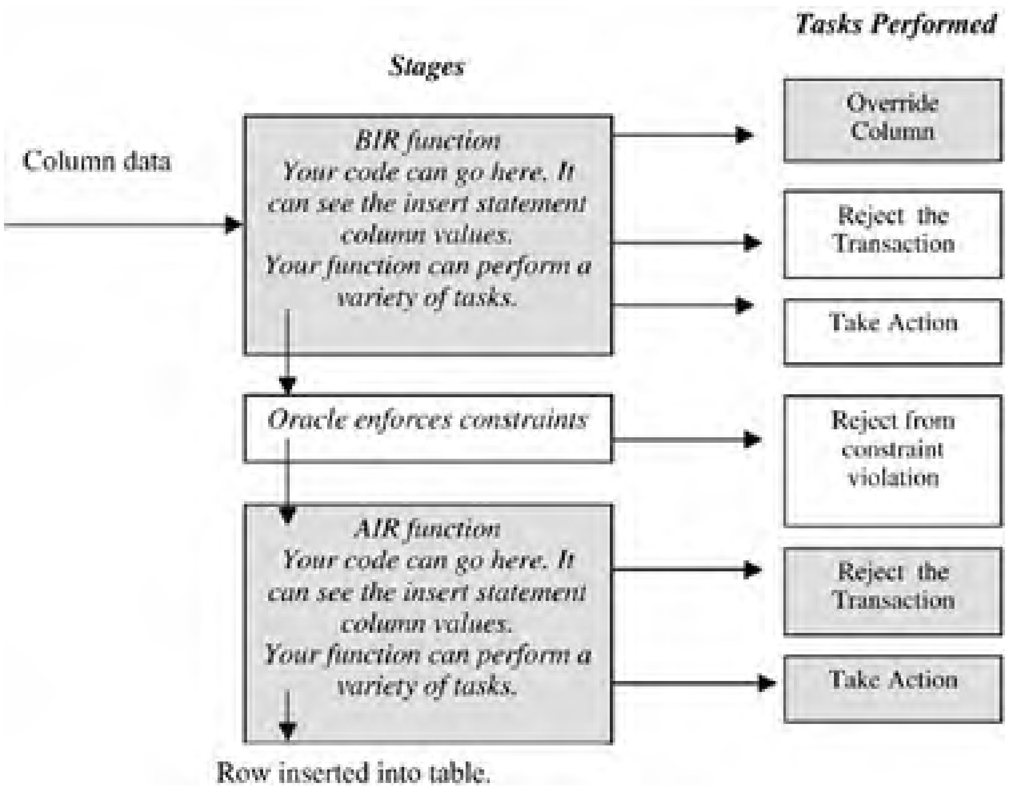
The trigger code first converts the :NEW values. This includes truncating the HIRE_DATE column. Following that, it checks for English department salary restriction. This can potentially reject the insert. Finally, the trigger takes the action to print.

```
CREATE OR REPLACE TRIGGER professors_bir
BEFORE INSERT ON professors
FOR EACH ROW
DECLARE
    msg VARCHAR2(100) :=
        'The salary exceeds the ENGL maximum of $10,000.00';
BEGIN
    -- -------------------------------------
    -- OVERRIDE COLUMN RULES
    -- -------------------------------------

    -- truncate hours, minutes, seconds of hire date
    :NEW.hire_date := TRUNC(:NEW.hire_date);

    -- round salary
    :NEW.salary := ROUND(:NEW.salary);

    -- convert department to upper case
    :NEW.department := UPPER(:NEW.department);
    -- -------------------------------------
    -- REJECT TRANSACTION RULES
    -- -------------------------------------
    IF :NEW.department='ENGL' AND :NEW.salary > 10000 THEN
        RAISE_APPLICATION_ERROR(-20000,msg);
    END IF;
    -- ---------------------------------------
    -- TAKE ACTION RULES
    -- ---------------------------------------
    dbms_output.put_line
       ('Before Insert Row Trigger Action');
END;
```

6.6.3 Before versus After

The aforementioned trigger enforces several business rules. All are enforced prior to Oracle's constraint enforcement. The "Override Column" trigger code must be in a Before-Insert Row trigger, but the "Reject the Transaction" and "Take Action" logic should be after the Oracle constraint. In general, you only want to process data for an application business rule that you know is good—data that has passed integrity checks. This shifts the location of the code to the architecture drawn in Figure 6-3.

Figure 6-3. Improved Location of Trigger Code.


Tasks Performed
Stages
Column data
BIR function
Your code can go here. It can see the insert statement column values.
Your function can perform a variety of tasks.
Override Column
Reject the Transaction
Take Action
Oracle enforces constraints
Reject from constraint violation
AIR function
Your code can go here. It can see the insert statement column values.
Your function can perform a variety of tasks.
Reject the Transaction
Take Action
Row inserted into table.

To implement this change we need to create two triggers and split the PL/SQL logic between the triggers. The BEFORE-INSERT trigger performs the "Override Column" function. The remaining trigger code goes in an AFTER-INSERT row trigger. The following trigger is the BEFORE-INSERT row trigger, with comments removed.

```
CREATE OR REPLACE TRIGGER professors_bir
BEFORE INSERT ON professors
FOR EACH ROW
BEGIN
    :NEW.hire_date := TRUNC(:NEW.hire_date);
    :NEW.salary := ROUND(:NEW.salary);
    :NEW.department := UPPER(:NEW.department);
END;
```

The AFTER-INSERT trigger, which runs after Oracle's constraint enforcement, will perform the "Reject the Transaction" task (check for an invalid salary condition) and perform the "Take Action" task, which is to print a message. The following is the AFTER-INSERT trigger, comments removed.

```
CREATE OR REPLACE TRIGGER professors_air
AFTER INSERT ON professors
FOR EACH ROW
DECLARE
    msg VARCHAR2(100) :=
        'The salary exceeds the ENGL maximum of'|| $10,000.00';
BEGIN
    IF :NEW.department='ENGL' AND :NEW.salary > 10000 THEN
        RAISE_APPLICATION_ERROR(-20000,msg);
    END IF;
    dbms_output.put_line
       ('Before Insert Row Trigger Action');
END;
```

6.6.4 Using Packages for Procedural Constraints

Rules for a trigger can pile up. A trigger can quickly become complex. Because an invalid trigger can interfere with the operations of an application, logic should be moved from the trigger to PL/SQL packages. If a piece of code is not performing as expected, that procedure call can be removed from the trigger. A trigger can and should be able to enforce many business rules, but each rule should be autonomous.

Each check constraint can be enabled and disabled with an ALTER TABLE statement. You should be able to accomplish the same level of granularity with triggers.

The following illustrates encapsulating the business rule logic in a package. This package is called PROFESSORS_CONS because it enforces constraints on the PROFESSORS table. The package specification is listed first and then the triggers.

```
CREATE OR REPLACE PACKAGE professors_cons IS

    FUNCTION salary(sal professors.salary%TYPE)
        RETURN professors.salary%TYPE;

    FUNCTION hire_date(hd professors.hire_date%TYPE)
        RETURN professors.hire_date%TYPE;

    FUNCTION department(dept professors.department%TYPE)
        RETURN professors.department%TYPE;

    PROCEDURE validate_salary
        (dept professors.department%TYPE,
         sal professors.salary%TYPE);

    PROCEDURE print_action;

END professors_cons;
```

The triggers only make calls to individual functions in the package. The BEFORE-INSERT trigger uses the logic in the package rather than perform each individual truncate and round. Granted this is not much logic to encapsulate. ROUND, UPPER, and TRUNC are not extensive data transformations. However, a more realistic business rule could stipulate that the HIRE_DATE on an insert be changed to the next business day. Such a transformation could include database lookups to determine the next working day.

The BEFORE-INSERT row trigger with calls to the constraints package is shown here.

```
CREATE OR REPLACE TRIGGER professors_bir
BEFORE INSERT ON professors
FOR EACH ROW
BEGIN
    :NEW.hire_date :=
            professors_cons.hire_date(:NEW.hire_date);
    :NEW.salary :=
            professors_cons.salary(:NEW.salary);
    :NEW.department :=
            professors_cons.department(:NEW.department);
END;
```

The AFTER-INSERT trigger is two procedure calls.

```
CREATE OR REPLACE TRIGGER professors_air
AFTER INSERT ON professors
FOR EACH ROW
BEGIN
    professors_cons.validate_salary
            (:NEW.department, :NEW.salary);
    professors_cons.print_action;
END;
```

The package body that implements the business rules is the following.

```
CREATE OR REPLACE PACKAGE BODY professors_cons IS

    FUNCTION salary(sal professors.salary%TYPE)
        RETURN professors.salary%TYPE IS
    BEGIN
        RETURN ROUND(sal);
    END salary;

    FUNCTION hire_date(hd professors.hire_date%TYPE)
        RETURN professors.hire_date%TYPE IS
    BEGIN
        RETURN TRUNC(hd);
    END hire_date;

    FUNCTION department(dept professors.department%TYPE)
        RETURN professors.department%TYPE IS
    BEGIN
        RETURN UPPER(dept);
    END department;

    PROCEDURE validate_salary
        (dept professors.department%TYPE,
         sal professors.salary%TYPE)
    IS
        msg VARCHAR2(100) :=
         'The salary exceeds the ENGL maximum of $10,000.00';
    BEGIN
        IF dept ='ENGL' AND sal  > 10000 THEN
            RAISE_APPLICATION_ERROR(-20000,msg);
        END IF;
    END validate_salary;

    PROCEDURE print_action IS
    BEGIN
        dbms_output.put_line
            ('Before Insert Row Trigger Action');
    END print_action;
END professors_cons;
```

6.6.5 Managing Error Codes and Messages

Large systems (e.g., operating systems and database products) do not embed error code numbers and messages as literal values within the code. The following procedure, taken from the professor's constraint enforcement package in Section 6.5, is an example of a buried error code number and message text.

```
PROCEDURE validate_salary
        (dept professors.department%TYPE,
         sal professors.salary%TYPE)
    IS
        msg VARCHAR2(100) :=
         'The salary exceeds the ENGL maximum of'|| '$10,000.00';
    BEGIN
        IF dept ='ENGL' AND sal  > 10000 THEN
            RAISE_APPLICATION_ERROR(-20000,msg);
        END IF;
    END validate_salary;
```

As new code is developed, the management of error codes is important. An application developer needs to know the next available message number. This can easily be determined when error code numbers are centralized. The following package is an example of encapsulating error messages.

The following errors package encapsulates the error numbers and message text for all the errors of an application.

```
 CREATE OR REPLACE PACKAGE errors IS
    eng_dept_sal CONSTANT PLS_INTEGER := -20001;
    app_error_02 CONSTANT PLS_INTEGER := -20002;
    app_error_03 CONSTANT PLS_INTEGER := -20003;

    eng_dept_sal_txt CONSTANT VARCHAR2(100) :=
        'The salary exceeds the ENGL maximum of'|| '$10,000.00';

    app_error_02_txt CONSTANT VARCHAR2(100) := 'vacant';
    app_error_03_txt CONSTANT VARCHAR2(100) := 'vacant';
END errors;
```

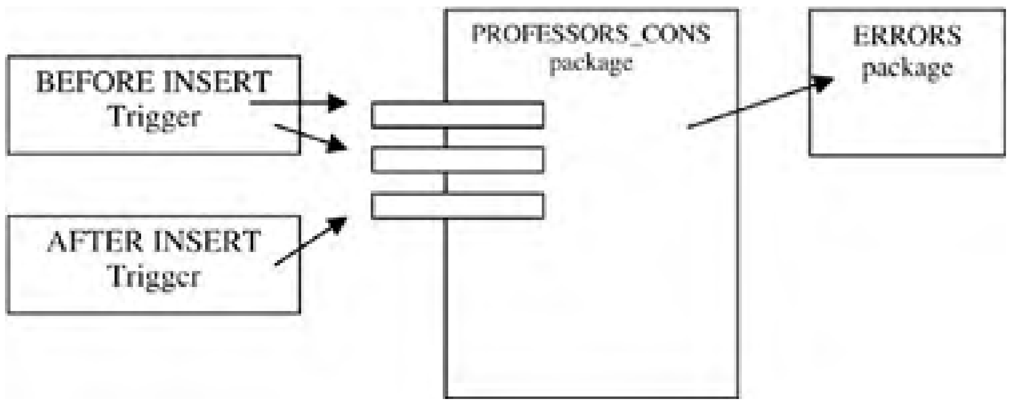
This changes the procedure in the constraints enforcement package for the professor's table to the following:

```
PROCEDURE validate_salary
        (dept professors.department%TYPE,
         sal professors.salary%TYPE)
    IS
    BEGIN
        IF dept ='ENGL' AND sal  > 10000 THEN
            RAISE_APPLICATION_ERROR
                (errors.eng_dept_sal,
                 errors.eng_dept_sal_txt);
        END IF;
    END validate_salary;
```

6.6.6 Trigger Architecture

We started with a set of business rules that were enforced within a single row trigger. Logically, it made sense to split the logic between the BEFORE-INSERT and AFTER-INSERT triggers. The specific logic of each rule was encapsulated into a package. The error definitions were further encapsulated into an errors package. The final architecture is shown in Figure 6-4.

Figure 6-4. Final Architecture.



The ERRORS package has no body. It is a central repository for constants that can be referenced by various parts of the business rule enforcement application code.

Chapter Seven. Statement Level Triggers

This chapter begins with the same trigger that started Chapter 6, except this is a statement level trigger. We start with a TEMP table.

```
DROP TABLE TEMP;
CREATE TABLE temp (N NUMBER);
```

Now create an ALTER-INSERT-STATEMENT trigger.

```
CREATE OR REPLACE TRIGGER temp_ais
AFTER INSERT ON TEMP
BEGIN
    dbms_output.put_line('executing temp_ais');
END;
```

The following inserts have different results than the beginning of Chapter 6, which demonstrated a ROW trigger. We begin by inserting a single row.

```
SQL> set feedback off
SQL> INSERT INTO temp VALUES (1);          -- insert 1 row
executing temp_ais
SQL> INSERT INTO temp VALUES (1);          -- insert 1 row
executing temp_ais
SQL> INSERT INTO temp SELECT * FROM temp;  -- insert 2 rows
executing temp_ais
```

The INSERT statement fires once per SQL statement. For the last insert a row trigger would have to fire twice.

7.1 Sequence of Events

You can do the following with an INSERT STATEMENT trigger:

- You can perform an aggregate computation on the table. This can be computed before or after the insert.
- You can use a statement level trigger to process data that is collected by a row trigger. This is covered in Section 7.4, "Processing Row Captured Data."
- You can signal an event. This can simply be a print statement. It can also be an email or signal to another process using the DBMS_ALERT package.

Figure 7-1 illustrates the behavior of statement level triggers that can perform "Take Action" functions of "Reject the Transaction" functions. The statement level trigger has no knowledge of the before and after values of any changed row. For a comparison with row triggers, compare Figure 7-1 with Figure 6-1.

Figure 7-1. Statement Level Trigger.

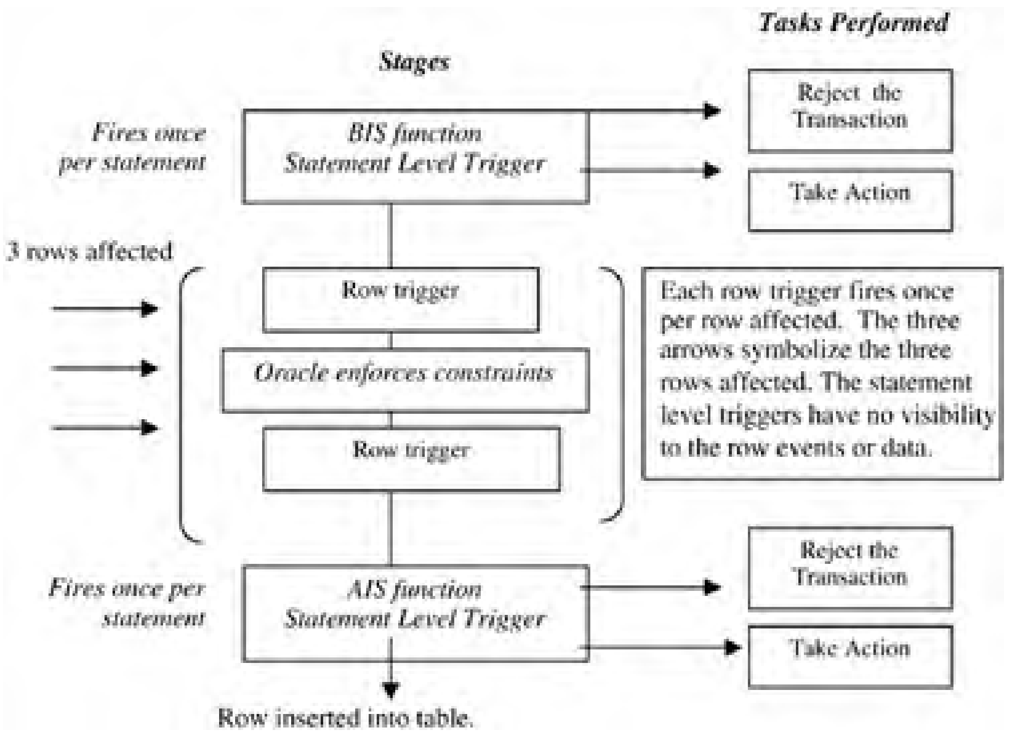

Tasks Performed
Stages
Reject the Transaction
Fires once per statement
BIS function Statement Level Trigger
Take Action
3 rows affected
Row trigger
Oracle enforces constraints
Row trigger
Each row trigger fires once per row affected. The three arrows symbolize the three rows affected. The statement level triggers have no visibility to the row events or data.
Reject the Transaction
Fires once per statement
AIS function Statement Level Trigger
Take Action
Row inserted into table.

Figure 7-1 illustrates the statement level behavior. It also shows that row level triggers fire between the before-statement and after-statement triggers. If three rows are affected by a SQL UPDATE, as is the example in this figure, three row triggers will fire. Statement level triggers fire once.

7.2 Insert Statement Trigger Syntax

The Insert Statement Trigger has the following syntax.

```
CREATE OR REPLACE TRIGGER trigger_name
[AFTER | BEFORE] INSERT ON table_name
DECLARE
    Local declarations
BEGIN
    Body written PL/SQL
END;
```

The key difference in the syntax between the statement and row trigger is the FOR EACH ROW clause—this clause specifically identifies the trigger as row level and is not in the statement level trigger.

The statement trigger syntax that designates the triggering event is the same as row triggers. Refer to Chapter 6, "Row Trigger Syntax," for a thorough discussion on trigger naming conventions and the OF COLUMN_NAME clause. The following are valid clauses for statement level triggers, as well as row triggers.

```
BEFORE INSERT OR UPDATE OR DELETE ON table_name

AFTER INSERT OR UPDATE OF column_name OR DELETE ON table_name
```

The following are two key points with regard to trigger options:

Table .

• WHEN (Boolean expression)	ALL ROW triggers only.
• OF column_name clause	Valid for UPDATE ROW and STATEMENT triggers only.

The syntax for statement level triggers is simpler than row triggers. The following features do not exist with statement level triggers:

- There is no WHEN (Boolean expression) that voids any discussion of OLD.COLUMN_NAME and NEW.COLUMN_NAME expressions.
- References to :NEW.COLUMN_NAME and :OLD.COLUMN_NAME are invalid.

The combination of row and statement triggers is 12 types of triggers. The following summarizes the template for each trigger.

Trigger Type	Triggering Event	Create or Replace (example: TEMP table)

BEFORE STATEMENT	INSERT	TRIGGER TEMP
		BEFORE INSERT ON TEMP
		BEGIN
		    Body
		END

BEFORE EACH ROW[a]	INSERT	TRIGGER TEMP
		BEFORE INSERT ON TEMP
		FOR EACH ROW
		BEGIN
		    Body
		END

AFTER EACH ROW[a]	INSERT	TRIGGER TEMP
		AFTER INSERT ON TEMP
		FOR EACH ROW
		BEGIN
		    Body
		END

AFTER STATEMENT	INSERT	TRIGGER TEMP

		AFTER INSERT ON TEMP
		BEGIN
		    Body
		END

BEFORE STATEMENT	UPDATE[b]	TRIGGER TEMP
		BEFORE UPDATE ON TEMP
		BEGIN
		    Body
		END

BEFORE EACH ROW[a]	UPDATE[b]	TRIGGER TEMP
		BEFORE UPDATE ON TEMP
		FOR EACH ROW
		BEGIN
		    Body
		END

AFTER EACH ROW[a]	UPDATE[b]	TRIGGER TEMP
		AFTER UPDATE ON TEMP
		FOR EACH ROW
		BEGIN
		    Body
		END

AFTER STATEMENT	UPDATE[b]	TRIGGER TEMP
		AFTER UPDATE ON TEMP
		BEGIN
		    Body
		END

BEFORE STATEMENT	DELETE	TRIGGER TEMP
		BEFORE DELETE ON TEMP
		BEGIN
		    Body
		END

BEFORE EACH ROW[a]	DELETE	TRIGGER TEMP
		BEFORE DELETE ON TEMP
		FOR EACH ROW
		BEGIN
		    Body
		END

AFTER EACH ROW[a]	DELETE	TRIGGER TEMP
		AFTER DELETE ON TEMP
		FOR EACH ROW
		BEGIN
		    Body
		END

AFTER STATEMENT	DELETE	TRIGGER TEMP
		AFTER DELETE ON TEMP
		BEGIN
		    Body
		END

[a] WHEN (Boolean expression) is optional with all row triggers.

[b] OF COLUMN_NAME clause is optional on UPDATE ROW and STATEMENT triggers.

Although this table illustrates 12 distinct triggers, any trigger type, such as a BEFORE STATEMENT trigger, can combine triggering events such as the following:

```
CREATE OR REPLACE TRIGGER temp_biuds
BEFORE INSERT OR UPDATE OR DELETE ON TEMP
BEGIN
    CASE
    WHEN inserting THEN
        PL/SQL code here
    WHEN updating THEN
        PL/SQL code here
    WHEN deleting THEN
        PL/SQL code here
    END CASE;
END;
```

7.3 Statement Level Aggregation

Statement triggers can enforce business rules where the rule is based on a table aggregate. Although a row trigger can restrain a salary for a particular row, a statement trigger can constrain the result of SUM(salary). This is essentially a constraint on the entire table.

This is quite different from constraints discussed so far. Initially, simple rules are enforced with a CHECK constraint. Chapter 6 illustrates a more complex rule that restricts the salary for any professor in the English department while leaving the CHECK constraint to enforce a salary restriction for non-English professors.

Consider the following rules.

> *A professor cannot be added if the current budget exceeds $55,000.*
>
> *The total budget cannot exceed $60,000.*

These can be enforced with statement level triggers. The first trigger, BEFORE STATEMENT, will reject any transaction that attempts to add a professor when the current budget exceeds $55,000. The AFTER STATEMENT trigger rejects the transaction when the result of adding a professor exceeds the sum of $60,000. The data for the PROFESSORS table in Chapter 4, Section 4.4, "Sample Data," shows the current sum at $50,000. We could add a single professor with a salary of $10,000. This would pass the business rule test. However, the BEFORE STATEMENT trigger will reject any additional inserts.

Secondly, we could insert a salary of $5,000. This would be allowed. However, the AFTER STATEMENT trigger will likely reject any sizable insert after that.

The implementation of these business rules will require the following steps.

- Update the ERRORS package with error numbers.
- Encapsulate the business rule logic in a constraints enforcement package.
- Write the before and after statement trigger.

The first step is to declare error numbers and message text in the errors package from Chapter 6. This is the central definition for errors raised through the application. The ERRORS package is updated to include two error numbers: –20002 and –20003.

```
CREATE OR REPLACE PACKAGE errors IS
  eng_dept_sal CONSTANT PLS_INTEGER := -20001;
  app_error_02 CONSTANT PLS_INTEGER := -20002;
  app_error_03 CONSTANT PLS_INTEGER := -20003;

  eng_dept_sal_txt CONSTANT VARCHAR2(100) :=
  'The salary exceeds the ENGL maximum of $10,000.00';

  app_error_02_txt CONSTANT VARCHAR2(100) :=
  'No additions if the budget exceeds $55,000.00';

  app_error_03_txt CONSTANT VARCHAR2(100) :=
  'Budget cannot be over $60,000.00';
END errors;
```

The next step is to encapsulate the business logic in a constraints package. Such a package was developed in Chapter 6. We can revive PROFESSORS_CONS. The following shows the package spec and body, minus the package procedure code from Chapter 6. The following code implements a single procedure that is used by the BEFORE and AFTER statement trigger. There are several styles for coding this. One is to make a separate procedure for each trigger. With this interface the trigger must pass the arguments that specify the constraint limit, error code, and text.

```
CREATE OR REPLACE PACKAGE professors_cons IS
  PROCEDURE constrain_budget
    (limit NUMBER,err_code PLS_INTEGER,err_text
    VARCHAR2);
END professors_cons;

CREATE OR REPLACE PACKAGE BODY professors_cons IS
  PROCEDURE constrain_budget
    (limit NUMBER,err_code PLS_INTEGER,err_text
    VARCHAR2)
  IS
    budget_sum NUMBER;
  BEGIN
    SELECT SUM(salary) INTO budget_sum FROM
    professors;
    IF budget_sum > limit THEN
        RAISE_APPLICATION_ERROR(err_code, err_text);
    END IF;
  END constrain_budget;
END professors_cons;
```

The BEFORE and AFTER statement triggers are the last and final step. As always, the body of the trigger is short and simple. Because both INSERT and UPDATE statements can potentially violate the rule, the triggering event is set to INSERT OR UPDATE.

```
CREATE OR REPLACE TRIGGER professors_bis
BEFORE INSERT OR UPDATE ON professors
BEGIN
    professors_cons.constrain_budget
        (55000, errors.budget_err_1,
        errors.budget_err_1_txt);
END;

CREATE OR REPLACE TRIGGER professors_ais
AFTER INSERT OR UPDATE ON professors
BEGIN
    professors_cons.constrain_budget
        (60000, errors.budget_err_2,
        errors.budget_err_2_txt);
END;
```

If these rules are in place (i.e., the packages and trigger compile) we can insert this next professor:

```
INSERT INTO professors VALUES
    ('Smith', 'Mathematics', SYSDATE,
      10000.00, 'YES','MATH');
```

However, any further INSERTs will be rejected by the BEFORE statement trigger. This is due to the fact that the current SUM(SALARY) exceeds $55,000.

We can insert this professor:

```
INSERT INTO professors VALUES
    ('Smith', 'Mathematics', SYSDATE,
      5000.00, 'YES','MATH');
```

However, any sizable addition to the staff will pass the first validation in the BEFORE statement trigger (the budget is at $55,000), but will be rejected by the second, the AFTER statement trigger. This would occur if the salary of this addition caused the SUM(SALARY) to exceed $60,000.

7.4 Processing Row Captured Data

Row triggers can store :OLD and :NEW column values in a global temporary table. The scope of a global temporary table is just that transaction. By copying :OLD and :NEW values, the processing of the business rule can be deferred to the statement level trigger. Sometimes this is necessary because the business rule is complex and requires queries from tables, including the table being updated.

The following illustrates the general technique. First a global temporary table is needed. This table will be used to store data in the row level trigger.

```
CREATE global temporary TABLE professors_g
 (prof_name     VARCHAR2(10),
  specialty     VARCHAR2(20),
  hire_date     DATE,
  salary        NUMBER(7,2),
  tenure        VARCHAR2(3),
  department    VARCHAR2(10)) ON COMMIT DELETE ROWS;
```

The next step is to code procedures in the constraints package for this table. These procedures will be added to the PROFESSORS_CONS package. Showing just the additions for the package specification:

```
CREATE OR REPLACE PACKAGE professors_cons IS
    PROCEDURE load_temp_table
        (v_prof_name  professors.prof_name%TYPE,
         v_specialty  professors.specialty%TYPE,
         v_hire_date  professors.hire_date%TYPE,
         v_salary     professors.salary%TYPE,
         v_tenure     professors.tenure%TYPE,
         v_department professors.department%TYPE);

    PROCEDURE dump_temp_table;

END professors_cons;
```

The package body is:

```
CREATE OR REPLACE PACKAGE BODY professors_cons IS

    PROCEDURE load_temp_table
        (v_prof_name  professors.prof_name%TYPE,
         v_specialty  professors.specialty%TYPE,
         v_hire_date  professors.hire_date%TYPE,
         v_salary     professors.salary%TYPE,
         v_tenure     professors.tenure%TYPE,
         v_department professors.department%TYPE)
    IS
    BEGIN
        INSERT INTO professors_g VALUES
            (v_prof_name, v_specialty, v_hire_date,
             v_salary, v_tenure, v_department);
    END load_temp_table;

    PROCEDURE dump_temp_table IS
    BEGIN
        FOR rec in (SELECT * FROM professors_g) LOOP
            dbms_output.put_line(
                rec.prof_name||' '||rec.specialty||' '||
                rec.hire_date||' '||rec.salary||' '||
                rec.tenure||' '||rec.department);
        END LOOP;
    END dump_temp_table;
END professors_cons;
```

The following is a AFTER DELETE ROW trigger. When this trigger fires it only inserts row data in the temporary table through the PROFESSORS_CONS package.

```
CREATE OR REPLACE TRIGGER professors_adr
AFTER DELETE ON professors
FOR EACH ROW
BEGIN
    professors_cons.load_temp_table
        (:old.prof_name, :old.specialty, :old.hire_date,
         :old.salary, :old.tenure, :old.department);
END;
```

The next trigger is an AFTER DELETE STATEMENT trigger that uses the constraints package to print the rows deleted. Although this demonstration merely prints the data, in some circumstances this can be useful to a statement level trigger. Statement level triggers have no knowledge of the rows affected by the SQL statement. They have no knowledge of :OLD and :NEW values.

The AFTER DELETE statement trigger is:

```
CREATE OR REPLACE TRIGGER professors_ads
AFTER DELETE ON professors
BEGIN
    professors_cons.dump_temp_table;
END;
```

The DELETE SQL statement is followed by the output from the statement trigger.

```
SQL> DELETE FROM professors;

Blake Mathematics 08-aug-2003 02:06:27 10000 YES MATH
Milton Am Hist 09-aug-2003 02:06:27 10000 YES HIST
Wilson English 06-aug-2003 02:06:27 10000 YES ENGL
Jones Euro Hist 12-jul-2003 02:06:28 10000 YES HIST
Crump Ancient Hist 12-jul-2003 02:06:28 10000 YES HIST

5 rows deleted.
```

◀ PREVIOUS NEXT ▶

Chapter Eight. Complex Rule Enforcement

The following scenario is based on the data in the STUDENT_VEHICLES table and the PARKING_TICKETS table. Refer to Chapter 4, Section 4.4, for the sample data.

A business rule can sometimes have a recursive nature. A general scenario is when an UPDATE statement executes and an update trigger modifies other rows in that same table. Deletes can also be recursive. Certainly, the foreign key delete cascade is one method of deleting dependent data. Other times, the business rule is complex and a delete trigger must procedurally determine if additional deletes are required.

The following scenario demonstrates a recursive delete. The delete row trigger may delete additional rows. It may not. It depends on the data. The business rule is:

> When a student pays a parking ticket, all other tickets for the same car and for the same amount are deleted as well, provided the sum of the tickets to be deleted does not exceed $10.00.

The rationale is that tickets are sometimes duplicated. The assumption is that multiple tickets for the same amount and the same car are accidental duplicates. If the sum of the tickets is excessive, then this is probably not the case. Hence the limit of $10.00.

A student can have three tickets, each $3.00. All are deleted when one ticket is paid. The data from the Parking Tickets table in Chapter 4 shows that there is one case of this. The vehicle with New York tag number MH 8709 has a duplicate parking violation. Each ticket is $5.00. According to this rule, one paid ticket deletes both parking tickets from the table.

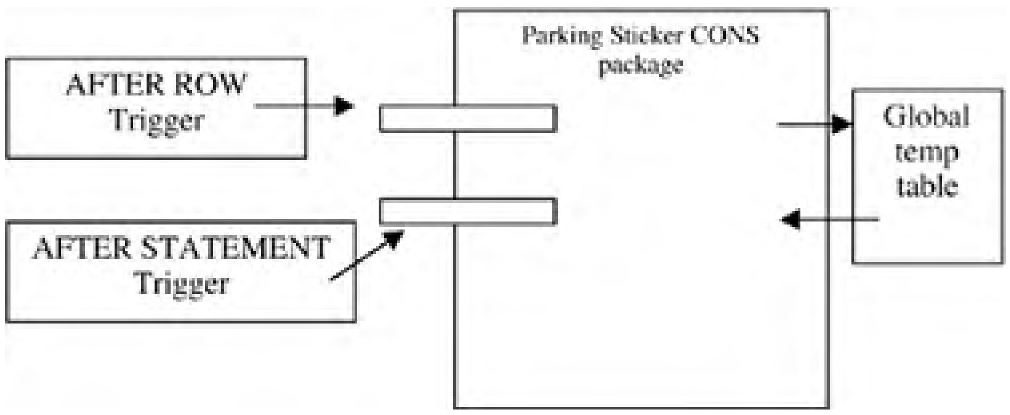
The task is to develop a trigger that takes the specific action to delete additional records within the same table. A ROW trigger cannot read or perform any DML operation on the table being modified. The reason is the table is being updated. Oracle guarantees consistent reads by queries read from rollback segments. The SELECT or any DML operation within the context of a row trigger generates an Oracle ORA-0491 Table Mutating error.

The approach in these situations is to capture row values that are needed to enforce the business rule and make them available to an AFTER statement trigger. Chapter 7 ended with an example similar to this. The following is a first step at that process. The package below is designed to capture the STATE, TAG_NO, and AMOUNT from the ROW trigger. This data is stored in a global temporary table. The following is the definition of the global temporary table. Upon a commit or rollback, data is cleared from the table.

```
CREATE GLOBAL TEMPORARY TABLE parking_tickets_g
(state  VARCHAR2(2),
 tag_no VARCHAR2(10),
 amount NUMBER) ON COMMIT DELETE ROWS;
```

The next step is a package that interfaces with the row and statement trigger, shown in Figure 8-1. The row trigger will pass values in a procedure call. Those values are inserted into the global temporary table. The statement level trigger calls a procedure that reads the row level data from the temporary table and enforces the business rule.

Figure 8-1. Parking Ticket Deletes Architecture.



The following is the specification and body of the trigger interface package. This is called the PARKING_TICKETS_CONS package. This code accepts row data from the row trigger. For statement level processing it simply prints the data read from the temporary table. This will be revised shortly to include the deletes.

```
CREATE OR REPLACE PACKAGE parking_tickets_cons IS

    PROCEDURE load_temp_table
        (v_state parking_tickets.state%TYPE,
         v_tag_no parking_tickets.tag_no%TYPE,
         v_amount parking_tickets.amount%TYPE);

    PROCEDURE remove_duplicates;
END parking_tickets_cons;
```

The package body is:

```
CREATE OR REPLACE PACKAGE BODY parking_tickets_cons IS
    PROCEDURE load_temp_table
        (v_state parking_tickets.state%TYPE,
         v_tag_no parking_tickets.tag_no%TYPE,
         v_amount parking_tickets.amount%TYPE)
    IS
    BEGIN
        INSERT INTO parking_tickets_g VALUES
            (v_state, v_tag_no, v_amount);
    END load_temp_table;

    PROCEDURE remove_duplicates IS
    BEGIN
        FOR rec in(SELECT * FROM parking_tickets_g) LOOP
            dbms_output.put_line
                ('REC:'||rec.state
                 ||' '||rec.tag_no||' '||rec.amount);
        END LOOP;
    END remove_duplicates;
END parking_tickets_cons;
```

The row trigger will call the aforementioned procedure LOAD_TEMP_TABLE. The statement level trigger will call REMOVE_DUPLICATES, which for now prints the rows selected from the temporary table.

The row and statement level triggers are:

```
CREATE OR REPLACE TRIGGER parking_tickets_adr
AFTER DELETE ON parking_tickets
FOR EACH ROW
BEGIN
    parking_tickets_cons.load_temp_table
        (:old.state, :old.tag_no, :old.amount);
END;

CREATE OR REPLACE TRIGGER parking_tickets_ads
AFTER DELETE ON parking_tickets
BEGIN
    parking_tickets_cons.remove_duplicates;
END;
```

If we execute the following SQL statement, the vehicle information is printed.

```
SQL> DELETE FROM parking_tickets
  2 WHERE ticket_no = ='P_02';

REC:NY MH 8709 5

1 row deleted.
```

The final step is to revise the logic in REMOVE_DUPLICATES. This procedure must query the PARKING_STUDENTS table to determine if there are other rows for the same vehicle and same ticket amount.

One row is already deleted. To determine if the sum of tickets is within the $10.00 range, the ticket amount (of the deleted row) must be added to the SUM(amount) left in the table. The revised REMOVE_DUPLICATES is the following:

```
PROCEDURE remove_duplicates
IS
    ct NUMBER;
BEGIN
    FOR rec in (SELECT * FROM parking_tickets_g) LOOP

        SELECT SUM(amount) INTO ct
        FROM   parking_tickets
        WHERE  state = rec.state
        AND    tag_no = rec.tag_no
        AND    amount = rec.amount;

        ct := ct + rec.amount;

        IF ct <= 10 THEN

            DELETE FROM parking_tickets
            WHERE state=rec.state AND tag_no=rec.tag_no;

        END IF;
    END LOOP;
END remove_duplicates;
```

Now the duplicates are removed. The following illustrates the single delete using the ticket number. The after statement trigger enforces an internal delete cascade based upon business logic.

```
SQL> SELECT * FROM parking_tickets;

TICKET_NO      AMOUNT ST TAG_NO
---------- ---------- -- ----------
P_01                5 CA CD 2348
P_02                5 NY MH 8709
P_03                5 NY MH 8709
P_04                5 NY JR 9837

SQL> DELETE FROM parking_tickets WHERE ticket_no='P_02';

SQL> SELECT * FROM parking_tickets;

TICKET_NO      AMOUNT ST TAG_NO
---------- ---------- -- ----------
P_01                5 CA CD 2348
P_04                5 NY JR 9837
```

The after statement trigger includes a DELETE statement on the table from which the original delete was applied.

Chapter Nine. The PL/SQL Environment

For traditional compiler languages, such as C, we write our source code and then compile it. We may choose from one of several compilers. The compilation produces a second file—an object file. We then use a linker to link one or more objects into an executable. The compile and link process leaves us with our source, several object files, and an executable image that is executed under the control of the operating system. The finished program may use some system library functions that perform services such as file open, put line, and file close.

The PL/SQL model is different and simpler from traditional programming, as just described. With PL/SQL, you start with writing your own source using a "SQL" filename extension. Once you have the ASCII text file with a "SQL" extension you compile it, but think of Oracle as your compiler. Oracle compiles your code and checks for syntax errors. After the compile, you have no additional files—there is no object file following the compile. The equivalent of the object and executable image is P-code that is stored in the database data dictionary. Additionally, Oracle stores the source code you just compiled in the data dictionary.

There are Interactive Development Environment (IDE) options to coding with a text editor, including Oracle Procedure Builder, which provides a graphical user interface (GUI) for creating, editing, and compiling both client- and server-side PL/SQL.

9.1 A Hello World Program

The text in this session includes screen ouput from SQL*Plus sessions. Refer to Chapter 2 for hints and use of SQL*Plus commands. The following table lists the common SQL*Plus commands used in this chapter.

SQL*Plus command	Description
SAVE filename	Saves the SQL*Plus buffer to a file with an SQL extension
@filename	Sends the file to Oracle for compile/execution. You do not need to type the ".SQL" extension.
SET SERVEROUTPUT ON	Sets up DBMS_OUTPUT to flush the buffer to the screen.
/	Sends the SQL*Plus buffer to Oracle for compile/execution.

From SQL*Plus, type the following Hello World program. First, type the SQL*Plus command SET SERVEROUTPUT ON, then the procedure text. The last line is a forward slash—this is SQL*Plus and instructs SQL*Plus to send the typed text to Oracle to be compiled.

```
SQL> SET SERVEROUTPUT ON
SQL> CREATE OR REPLACE PROCEDURE hello IS
  2  BEGIN
  3      DBMS_OUTPUT.PUT_LINE('Hello');
  4  END;
  5  /

Procedure created.

SQL> execute hello
Hello

PL/SQL procedure successfully completed.
```

Use the DBMS_OUTPUT package for standard output. The DBMS_OUTPUT package has overloaded procedures for different datatypes: DATE, NUMBER, and VARCHAR. The package specification for the PUT_LINE procedure is

```
procedure put_line (arg VARCHAR2);
procedure put_line (arg NUMBER);
procedure put_line (arg DATE);
```

You must execute the SQL*Plus command once for each session if you use DBMS_OUTPUT.

```
SET SERVEROUTPUT ON
```

DBMS_OUTPUT buffers output to a session-specific DBMS_OUTPUT buffer. Each call to the PUT_LINE procedure does not immediately dump text to your screen—it stays in the buffer. The SET SERVEROUTPUT ON command directs the SQL*Plus session to dump buffered text to your screen upon completion of a program.

The default buffer size is 20,000 characters. You can increase this with a call to the ENABLE procedure in DBMS_OUTPUT. The maximum is 1,000,000 characters.

```
SQL> execute dbms_output.enable(1000000);

PL/SQL procedure successfully completed.
```

The content of the SQL*Plus buffer is currently the Hello program. List the SQL*Plus buffer (lower case L) and save (SAVE) it to a file.

```
SQL> l
  1  CREATE OR REPLACE PROCEDURE hello IS
  2  BEGIN
  3      dbms_output.put_line('Hello');
  4* END;
SQL> save hello
```

```
Created file hello
SQL>
```

You just saved the contents of the SQL*Plus buffer to a host file HELLO.SQL. Run the script. This recompiles the HELLO procedure.

```
SQL> @hello.sql

Procedure created.
```

If you try to create a table and it exists, an error comes back. To recreate a table you must first drop the table and then create it. This prevents accidental recreating of a table with vital data. Stored procedures have a CREATE OR REPLACE syntax that is consistent with most programming languages. This syntax creates the procedure if it does not exist; if it does exist, Oracle recompiles it.

Edit the file HELLO.SQL and insert invalid syntax. Change PUT_LINE to PUTLINE. Add two additional lines at the end. Add a SQL*Plus command, SHOW ERRORS; then add the SQL*Plus LIST (L) command. The edited file, seven lines long, is the following:

```
CREATE OR REPLACE PROCEDURE hello IS
BEGIN
    dbms_output.putline('Hello');
END;
/
show errors
l
```

Run the command file with @HELLO. This will (a) compile the procedure with errors; (b) execute the SQL*Plus SHOW ERRORS command, which will list the offending line of PL/SQL code; and (c) provide a full listing of the procedure just compiled with the SQL*Plus LIST command.

```
SQL> @hello

Warning: Procedure created with compilation errors.

Errors for PROCEDURE HELLO:

LINE/COL ERROR
-------- -----------------------------------------------
3/5      PL/SQL: Statement ignored
3/17     PLS-00302: component 'PUTLINE' must be declared
  1  CREATE OR REPLACE PROCEDURE hello IS
  2  BEGIN
  3      dbms_output.putline('Hello');
  4* END;
```

Correct the host file, HELLO.SQL, and replace PUTLINE with PUT_LINE. Compile and execute. Suppress SQL*Plus feedback messages with SET FEEDBACK OFF:

```
SQL> set feedback off
SQL> @hello
No errors.
SQL> execute hello
Hello
```

Create a PL/SQL block that invokes the HELLO procedure. Do this by creating a text file with a SQL extension. Name the file RUN_HELLO.SQL. The following illustrates the syntax for a PL/SQL block (the forward slash is the SQL*Plus command to compile and execute the script).

```
DECLARE
    Variables
BEGIN
    Body of PL/SQL code.
END;
/
```

The DECLARE part is optional; it is not necessary if the block uses no variables.

A PL/SQL block is different from a stored procedure. HELLO is a compiled object in the database. You must first compile HELLO.SQL; then you can execute the procedure.

The text for RUN_HELLO.SQL, shown next, is seven lines long and includes a comment and a forward slash in the last line. This is a PL/SQL block. It executes the HELLO procedure five times.

```
-- Filename: RUN_HELLO.SQL
BEGIN
    FOR run_count IN 1..5 LOOP
        hello;
    END LOOP;
END;
/
```

A single command to compile and execute a PL/SQL block is:

```
@filename    -- the SQL file extension is not necessary
```

Building the HELLO procedure includes two steps:

1. First, you compile HELLO.SQL—this step validates the language syntax and compiles the source in the database.
2. The second step is to execute the procedure.

You build the PL/SQL block with one step: compile-and-execute. To run the PL/SQL block in SQL*Plus:

```
SQL> @run_hello
Hello
Hello
Hello
Hello
Hello
```

Use PL/SQL blocks as test drivers for stored procedures. There is nothing different about the code in a PL/SQL block and the code in stored procedure—both use PL/SQL. Enhance test driver code with exception handling code to display an exception error number and error message. The PL/SQL block below includes a WHEN OTHERS exception handler that prints the exception to a duplicate insert. First, create a table with a primary key constraint.

```
CREATE TABLE TEMP(N NUMBER CONSTRAINT PK_TEMP PRIMARY KEY);
```

TEST_TEMP.SQL is the name of the PL/SQL block and includes an exception handler. In contains 11 lines, including an initial comment and a forward slash as the last line, which must be in Column 1.

```
-- Filename: TEST_TEMP.SQL
BEGIN
    INSERT INTO temp VALUES (1);
    INSERT INTO temp VALUES (1);
EXCEPTION
    WHEN OTHERS THEN
        dbms_output.put_line('Error code:'||SQLCODE||'***');
        dbms_output.put_line
            ('Error message:'||SQLERRM||'***');
END;
/
```

In this PL/SQL block, the second insert fails with a primary key constraint violation. The code in the exception handler uses DBMS_OUTPUT to print the error number and message.

```
SQL> @test_temp
Error code:-1***
Error message:ORA-00001: unique constraint (SCOTT.PK_TEMP)
violated***
```

PL/SQL blocks convert to stored procedures by adding CREATE OR REPLACE. The following procedure, TEST_TEMP, implements the preceding script as a compiled procedure in the database. The following includes a corrected INSERT statement that will not violate the primary key constraint.

```
-- Filename: TEST_TEMP.SQL
CREATE OR REPLACE PROCEDURE TEST_TEMP IS
BEGIN
    INSERT INTO temp VALUES (1);
    INSERT INTO temp VALUES (2);
EXCEPTION
    WHEN OTHERS THEN
        dbms_output.put_line('Error code:'||SQLCODE||'***');
        dbms_output.put_line
            ('Error message:'||SQLERRM||'***');
END;
/
```

9.2 Referencing Oracle Packages

Each language has syntax for referencing other program units. The C programming language uses an INCLUDE directive to reference header files.

```
#include <stdio.h>
#include <stdlib.h>
```

Perl scripts call procedures in the Windows32 library with a USE WIN32 directive.

```
use Win32::Registry;
use Win32::OLE;
```

Java classes import whole packages and package classes.

```
package project.students;
import java.io.*;
import java.util.*;
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.*;
import project.util.*;
```

With each of these environments, it is a task to locate the library code you need and make sure it is in your path. PL/SQL does not require a compiler directive to use other packages. If you want to write PL/SQL that uses the DBMS_OUTPUT package, or another Oracle PL/SQL package, you simply write the procedure call in your code.

9.2.1 Creation of the Environment

The database administrator compiles many PL/SQL packages into the Oracle SYS account. This is part of creating the database. The database creation process also creates public synonyms and public grants for these packages. Naturally, some packages are intended for Oracle internal use and have restricted access.

These packages provide a robust Application Programming Interface (API). You can use the Oracle API to develop PL/SQL procedures that use methods in Java classes, write to host files, send mail through an SMTP service, and many other functions.

As you write PL/SQL you display output using the DBMS_OUTPUT package. This is one component of the API, and for most programmers, the most frequently used package during development. The API for DBMS_OUTPUT is listed in Section 9.6.

The following illustrates the SYS statements executed, during database creation, that make DBMS_OUTPUT available for general use.

```
1.GRANT EXECUTE ON DBMS_OUTPUT TO PUBLIC;
2.CREATE PUBLIC SYNONYM DBMS_OUTPUT FOR SYS.DBMS_OUTPUT;
```

The first statement means that SCOTT and all future Oracle accounts can write PL/SQL that use the DBMS_OUTPUT package. The keyword PUBLIC gives the grant to all users. The second statement means that SCOTT can write PL/SQL with statements like:

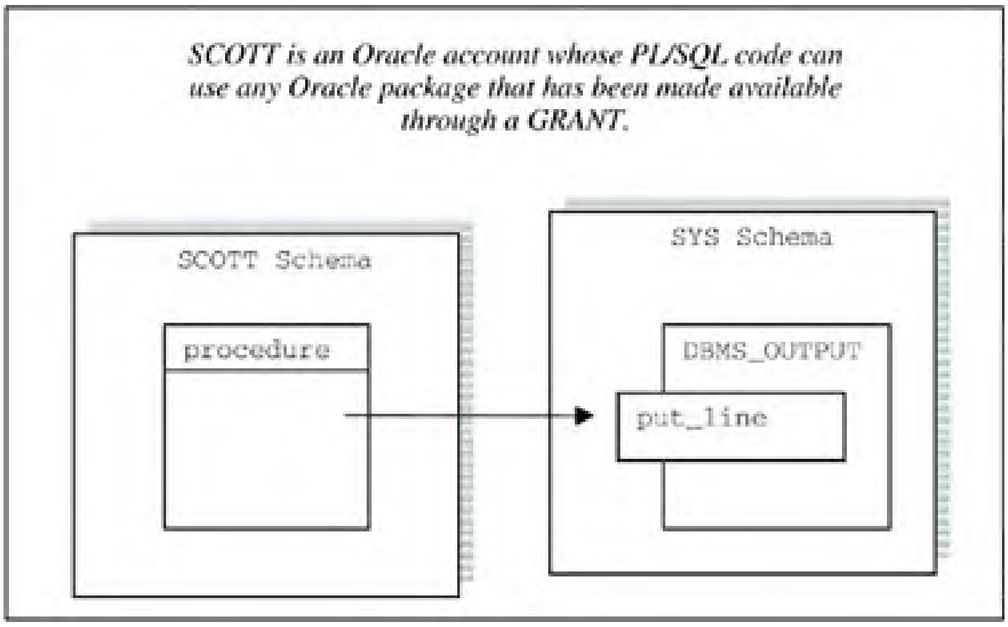
```
DBMS_OUTPUT.PUT_LINE('Hello');
```

Without the second statement, SCOTT would code the following:

```
SYS.DBMS_OUTPUT.PUT_LINE('Hello');
```

Figure 9-1 illustrates the encapsulation of packages in the SYS account.

Figure 9-1. Referencing the Oracle PL/SQL Packages.


SCOTT is an Oracle account whose PL/SQL code can use any Oracle package that has been made available through a GRANT.
SCOTT Schema
procedure
SYS Schema
DBMS_OUTPUT
put_line

When SCOTT compiles a procedure, Oracle identifies all referenced objects. If the SCOTT procedure references the DBMS_OUTPUT package, Oracle determines that DBMS_OUTPUT is a synonym for SYS.DBMS_OUTPUT and that SCOTT has EXECUTE privileges on that package.

9.2.2 The API

How extensive is the API? It includes hundreds of packages. Most packages begin with DBMS. Some utility packages begin with UTL such as UTL_SMTP—a PL/SQL API interface to the SMTP service.

To preview all packages in the API that begin with DBMS or UTL, query the data dictionary view ALL_OBJECTS. The following SQL generates a spool file listing the DBMS and UTL packages. This script runs with TERM off, which turns terminal-output off. The script directs output strictly to the spool file, ALL_OBJECTS.LST.

```
-- Filename ALL_OBJECTS.SQL
set pagesize 0
set term off
set feedback off
spool all_objects
SELECT  object_name
FROM    all_objects
WHERE  owner='SYS' AND
       object_type='PACKAGE'
AND    (object_name like 'DBMS%' OR object_name like 'UTL%');
set feedback on
set term on
spool off
```

The output file ALL_OBJECTS.LST will include hundreds of packages. The following illustrates the text of the list file generated:

```
SQL> @ALL_OBJECTS
This shows a few of the hundreds of packages.
DBMS_ALERT
DBMS_APPLICATION_INFO
DBMS_AQ
DBMS_AQADM
DBMS_AQADM_SYS
DBMS_AQADM_SYSCALLS
DBMS_AQIN
DBMS_AQJMS
DBMS_AQ_EXP_HISTORY_TABLES
```

As with any API, how do you use it? There are three approaches to learning the API for an Oracle package.

- You can use the SQL*Plus Describe command. This is a definition of the interface only.
- You can extract the package source from the data dictionary. This frequently includes examples and detailed descriptions on using the API.
- You can visit the free Oracle Web site, *technet.oracle.com*, to review the package documentation.

You can describe a package specification with the SQL*Plus DESCRIBE command. Examples in the text use the DESCRIBE command for tables. This command also describes views procedures, functions, and packages. The DESCRIBE command shows the interface. This includes procedure and function names with each parameter type, mode, and default option. This is useful if you are already familiar with an API and need to review the specifics of the interface.

```
SQL> DESC name-of-package-procedure-function
```

For example, the following spools the interface specification for the DBMS_OUTPUT package to a file, DBMS_OUTPUT_SPEC.LST. The following session output shows a partial listing—there are many other procedures to the DBMS_OUTPUT package.

```
SQL> spool dbms_output_spec
SQL> desc dbms_output

PROCEDURE ENABLE
  Argument Name     Type         In/Out Default?
  ------------------ --------------- ------ --------
  BUFFER_SIZE       NUMBER(38)     IN    DEFAULT

PROCEDURE PUT_LINE
  Argument Name     Type         In/Out Default?
  ------------------ --------------- ------ --------
  A             VARCHAR2      IN

SQL> spool off
```

The SQL*Plus DESCRIBE output of a package is a tabulated style view of the package specification. This may not be sufficient information if you are learning to use this package for the first time. It is merely a definition of the interface at a quick glance. The preceding DESCRIBE output shows procedures ENABLE and PUT_LINE, which have this interface definition.

```
procedure enable(buffer_size IN NUMBER);
procedure PUT_LINE(A VARCHAR2);
```

You can describe functions and procedures as well. Refer to Section 9.1, "DBMS_OUTPUT," for additional information on using DBMS_OUTPUT.

A second option is to pull the package specification source from the data dictionary with a query against the view ALL_SOURCE. The result is the package interface specification that frequently includes comments on how to use the API. The SQL to perform this is included in Section 9.6, "USER_SOURCE."

9.3 USER_OBJECTS

The USER_OBJECTS view provides status information on objects you create. This includes tables, sequences, views, stored procedures, database links, and others. The following is a partial description of columns from this view. Use this view to determine if a stored procedure is valid, if you need to recompile it, or to determine its last compile timestamp.

Refer to Chapter 5 for a complete description of the differences between the USER, ALL, and DBA data dictionary views.

- USER_OBJECTS provides information only on those objects you have created in your account.
- ALL_OBJECTS provides information on objects you have created plus objects to which you have privileges.
- DBA_OBJECTS provides information on all objects in the database. You need the Oracle role DBA or SELECT_CATALOG_ROLE to access DBA views.

Because the scope of DBA views is everything in the database, you must have either the Oracle DBA role or the Oracle SELECT_CATALOG_ROLE role. The DBA role has high privileges. SELECT_CATALOG_ROLE is intended for users who need to query data dictionary views. Application developers should be given this role.

A procedure you create will have an entry in USER_OBJECTS. If BLAKE creates a procedure HELLO_BLAKE and grants execute on that procedure to you, then you can see this object when you query OWNER, OBJECT_NAME, and OBJECT_TYPE from ALL_OBJECTS.

```
SQL> desc user_objects
 Name                          Null?    Type
 ----------------------------- -------- ---------------
 OBJECT_NAME                            VARCHAR2(128)
 OBJECT_TYPE                            VARCHAR2(18)
 CREATED                                DATE
 LAST_DDL_TIME                          DATE
 STATUS                                 VARCHAR2(7)
 And other columns
```

These are the columns relevant to object name, type, and status. There are a few other columns such as the OBJECT_ID of the object in the database. Relevant to this discussion, these columns have the following meaning.

OBJECT_NAME	This is the name in the CREATE OR REPLACE clause. This is not the host file. Running the script @MY_HELLO.SQL with a CREATE OR REPLACE PROCEDURE HELLO statement creates the object name HELLO. The data dictionary stores all attributes in upper case.
OBJECT_TYPE	For PL/SQL this is FUNCTION, PROCEDURE, PACKAGE, or PACKAGE BODY. There is never an underscore in PACKAGE BODY.

CREATED	This is the program creation date. To see the precise time of creation, change the default display for a DATE format:
	`ALTER SESSION SET`
	`NLS_DATE_FORMAT = 'DD-MON HH24:MI';`
	If you drop a procedure, running the CREATE OR REPLACE script resets the CREATED date. This is equivalent to creating the object for the first time. Once created, you can recompile it many times. The recompile data is the LAST_DDL_TIME date.

LAST_DDL_TIME	You can recompile a stored procedure many times. The recompile might fail or it might succeed. Either way, the recompilation updates the column LAST_DDL_TIME.
STATUS	Objects can have a status of VALID or INVALID. A procedure that inserts into the STUDENTS table becomes invalid if you drop the STUDENTS table. All objects should be VALID.

The following SQL is a general report on all the PL/SQL program units you have compiled into your schema.

```
-- Filename CHECK_PLSQL_OBJECTS
column object_name format a20
column last_ddl_time heading last_ddl
SELECT object_name||
       decode(object_type,'PROCEDURE','(P)',
       'FUNCTION','(F)', 'PACKAGE','(Spec)',
       'PACKAGE BODY','(Body)') object_name,
       status, created, last_ddl_time
FROM   user_objects
WHERE  object_type in ('PROCEDURE', 'FUNCTION',
       'PACKAGE','PACKAGE BODY');
```

This output includes each procedure, function, specification and body, the status, and last compile time. If there is one procedure, HELLO, the result is:

```
OBJECT_NAME    STATUS  CREATED           LAST_DDL_TIME
-------------- ------- ----------------- -----------------
HELLO(P)       VALID   14-jul-2003 16:18 14-jul-2003 16:18
```

The SQL built-in function USER evaluates to your current Oracle session account. To see what packages other users have extended to you, excluding the data dictionary SYS packages, select all PACKAGE type objects from ALL_OBJECTS and exclude SYS and yourself:

```
SELECT owner, object_name FROM all_objects WHERE object_type='PACKAGE' AND owner <> ' SYS'
AND owner <> USER;
```

9.4 Dependencies among Procedures

This section covers the following topics:

- Editing and compiling a procedure that invalidates other procedures.
- LAST_DDL_TIME and STATUS from USER_OBJECTS.
- Compiling a schema with DBMS_UTILITY.COMPILE_SCHEMA.
- Recompiling procedures, functions, and packages individually.
- This section uses the script CHECK_PLSQL_OBJECTS from the previous section.

When we first compile the HELLO procedure, the CREATED time and LAST_DDL_TIME are identical.

```
OBJECT_NAME    STATUS  CREATED           LAST_DDL_TIME
-------------- ------- ----------------- -----------------
HELLO(P)       VALID   14-jul-2003 16:18 14-jul-2003 16:18
```

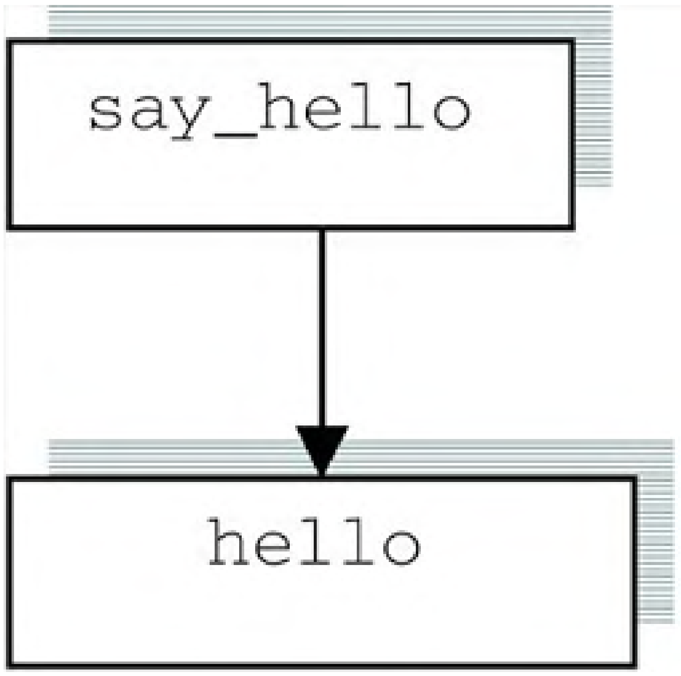
If we attempt to recompile the procedure and the compile fails, the procedure is still in the data dictionary but with an INVALID status. The LAST_DDL_TIME reflects the last compile time.

Executing a procedure that is INVALID will fail with an Oracle error:

```
PLS-00905: object SCOTT.HELLO is invalid
```

If Procedure A calls Procedure B and B becomes invalid, then A automatically becomes invalid. For the Figure 9-2 procedure, SAY_HELLO calls HELLO. What happens if HELLO becomes invalid?

Figure 9-2. Simple Procedure Dependency.



We begin with the code to HELLO.SQL and SAY_HELLO.SQL.

```
-- Filename HELLO.SQL
CREATE OR REPLACE PROCEDURE hello IS
BEGIN
    dbms_output.put_line('Hello');
END;
/
show errors

-- Filename SAY_HELLO.SQL
CREATE OR REPLACE PROCEDURE say_hello IS
BEGIN
    hello;
END;
/
show errors
```

Compile procedures HELLO and SAY_HELLO in order. The SHOW ERRORS command reports any compilation errors. The script CHECK_PLSQL_OBJECTS shows the STATUS as VALID for each procedure in USER_OBJECTS.

```
SQL> @CHECK_PLSQL_OBJECTS

OBJECT_NAME          STATUS  CREATED      LAST_DDL_TIM
-------------------- ------- ------------ ------------
HELLO(P)             VALID   25-jul 12:52 25-jul 01:02
SAY_HELLO(P)         VALID   25-jul 01:01 25-jul 01:02
```

Edit HELLO.SQL and change PUT_LINE to PUTLINE. The procedure will now compile with an error. Recompile HELLO with @HELLO.SQL. The status of SAY_HELLO is also invalid, yet we did not change the procedure. SAY_HELLO depends on a valid HELLO procedure. A compile error in Hello resulted in Oracle searching objects that depend on HELLO and invalidating those objects. All dependents of any procedure must be valid for that procedure to be valid, showing both objects as invalid:

```
SQL> @CHECK_PLSQL_OBJECTS

OBJECT_NAME          STATUS  CREATED      LAST_DDL_TIM
-------------------- ------- ------------ ------------
HELLO(P)             INVALID 25-jul 12:52 25-jul 01:05
SAY_HELLO(P)         INVALID 25-jul 01:01 25-jul 01:02
```

Correct the PL/SQL code in HELLO.SQL and recompile. The HELLO should be valid with a successful recompilation. What about SAY_HELLO, is this still invalid?

```
SQL> @hello

Procedure created.

SQL> @CHECK_PLSQL_OBJECTS

OBJECT_NAME          STATUS  CREATED      LAST_DDL_TIM
-------------------- ------- ------------ ------------
HELLO(P)             VALID   25-jul 12:52 25-jul 01:17
SAY_HELLO(P)         INVALID 25-jul 01:01 25-jul 01:02
```

Procedure SAY_HELLO is still invalid; however, when we execute the procedure, Oracle sees that it is invalid and attempts to validate it. This will be successful because all dependents (i.e., the HELLO procedure) are valid. Oracle compiles SAY_HELLO, sets the status to valid, and then executes the procedure. Following execution of HELLO, both procedures are valid.

```
SQL> execute say_hello
Hello

PL/SQL procedure successfully completed.

SQL> @CHECK_PLSQL_OBJECTS

OBJECT_NAME          STATUS  CREATED      LAST_DDL_TIM
-------------------- ------- ------------ ------------
HELLO(P)             VALID   25-jul 12:52 25-jul 01:17
SAY_HELLO(P)         VALID   25-jul 01:01 25-jul 01:17
```

There is understandable overhead with Oracle attempting to validate objects at run time. If HELLO is a widely used procedure and becomes invalid, there will be some performance degradation. During the normal operations of an application, Oracle may encounter many packages that became invalid and recompile them at run time. This can cause a noticeable impact to end users.

The following discussion covers the scenario when invalid code does not recompile.

We invalidated HELLO, recompiled it, and it became valid again. The change was a simple statement change that we corrected. A major code change to HELLO could cause recompilation failures in other procedures. Such an event would occur if the interface to HELLO changed. Changing the parameter specification, parameter types, or parameter modes can permanently invalidate other code.

If we change a procedure and recompile, Oracle's recompilation of other procedures may fail. Why wait until run-time to realize there is broken code. When PL/SQL changes occur, you can recompile the entire suite of PL/SQL code in the schema. The Oracle DBMS_UTILITY package provides this functionality with the COMPILE_SCHAME procedure.

To recompile all PL/SQL in a schema (this example uses the schema name, SCOTT):

```
SQL> execute dbms_utility.compile_schema('SCOTT')

PL/SQL procedure successfully completed.
```

The response "procedure successfully completed" means the call to DBMS_UTILITY was successful. There may be invalid objects. Run CHECK_PLSQL_OBJECTS for invalid stored procedures. LAST_DDL_TIME shows the recompilation time of each procedure.

If a procedure is invalid, you can SHOW ERRORS on that procedure, showing why it failed to compile with the following:

- SHOW ERRORS <type> <schema>.<name>
- SHOW ERRORS PROCEDURE *procedure_name;*
- SHOW ERRORS FUNCTION *function_name;*
- SHOW ERRORS PACKAGE *package_name;* (package spec errors)
- SHOW ERRORS PACKAGE BODY package_name; (package body errors)

To show compile errors for SAY_HELLO:

```
SHOW ERRORS PROCEDURE SAY_HELLO;
```

The following scenario includes three procedures. P1 calls P2, which calls P3. The procedure code is:

```
CREATE OR REPLACE procedure P3 IS
BEGIN
   dbms_output.put_line('executing p3');
END;
/
CREATE OR REPLACE procedure P2 IS
BEGIN
   P3;
END;
/
CREATE OR REPLACE procedure P1 IS
BEGIN
   P2;
END;
/
```

Compile these procedures in the following order: P3, then P2, then P1. Execution of P1 produces the following:

```
SQL> execute p1
executing p3
```

Change P3 by adding a parameter to the interface and compile the procedure.

```
CREATE OR REPLACE procedure P3(N INTEGER) IS
BEGIN
    dbms_output.put_line('executing p3');
END;
/
```

Not knowing all the dependencies on P3, we can compile the schema.

```
SQL> execute dbms_utility.compile_schema('SCOTT');

PL/SQL procedure successfully completed.
```

Check for invalid objects.

```
SQL> @check_plsql_objects

OBJECT_NAME          STATUS  CREATED      last_ddl
-------------------- ------- ------------ ------------
P1(P)                INVALID 25-jul 15:26 25-jul 15:35
P2(P)                INVALID 25-jul 15:26 25-jul 15:35
P3(P)                VALID   25-jul 15:26 25-jul 15:35
```

We have invalid objects P1 and P2. Use SHOW ERRORS to see why these procedures failed to compile.

```
SQL> show errors procedure p1
Errors for PROCEDURE P1:

LINE/COL ERROR
-------- ----------------------------------------------------
3/5      PLS-00905: object SCOTT.P2 is invalid
3/5      PL/SQL: Statement ignored

SQL> show errors procedure p2
Errors for PROCEDURE P2:

LINE/COL ERROR
-------- -----------------------------------------------------
3/5      PLS-00306: wrong number or types of arguments
         in call to 'P3'
3/5      PL/SQL: Statement ignored
```

Many invalid objects can pose a challenging problem. In the preceding example (P1, P2 and P3), there are two invalid objects. We changed P3 and saw from SHOW ERRORS that P2 is passing the wrong number of arguments to P3.

The DBMS_UTILITY.COMPILE_SCHEMA compiles all the PL/SQL in a schema. You can validate individual components with the ALTER statement.

```
ALTER PROCEDURE procedure_name COMPILE;

ALTER FUNCTION function_name COMPILE
```

To compile the package specification and body:

```
ALTER PACKAGE package_name COMPILE;
```

To compile just the package specification:

```
ALTER PACKAGE package_name COMPILE SPECIFICATION;
```

To compile just the package body:

```
ALTER PACKAGE package_name COMPILE BODY;
```

You can always determine object dependencies by querying the data dictionary view USER_DEPENDENCIES, covered in the next section. The preceding scenario includes three procedures: P1, P2, and P3. This is not a complex architecture. When there are many program units with many dependencies, the task becomes tedious. It requires repeated queries of the USER_DEPENDENCIES view. In the next section, we look at applying a general solution to querying USER_DEPENDENCIES.

The script used to query the USER_OBJECTS view, CHECK_PLSQL_OBJECTS, filters stored procedures with a WHERE clause. This script filters stored procedures just to demonstrate an example. A change to a stored procedure can invalidate other object types. A view can use a PL/SQL function. A trigger can use a procedure, function, or package. Objects from other schemas may use our PL/SQL objects. A general dependency tracing strategy requires that you query the ALL_DEPENDENCIES view for all object types.

9.5 USER_DEPENDENCIES

The view USER_DEPENDENCIES shows all dependencies for objects including procedures, functions, package specifications, and package bodies. Query this view to see all PL/SQL procedures that are dependent on a table you are about to drop. The description is:

```
SQL> desc user_dependencies
 Name                             Null?    Type
 ------------------------------ -------- ---------------
 NAME                             NOT NULL VARCHAR2(30)
 TYPE                                      VARCHAR2(12)
 REFERENCED_OWNER                          VARCHAR2(30)
 REFERENCED_NAME                           VARCHAR2(64)
 REFERENCED_TYPE                           VARCHAR2(12)
 REFERENCED_LINK_NAME                      VARCHAR2(128)
 SCHEMAID                                  NUMBER
 DEPENDENCY_TYPE                           VARCHAR2(4)
```

If you write a procedure, HELLO, that inserts a row in the STUDENTS table, there will be a row in this view with NAME=HELLO and REFERENCED_NAME=STUDENTS.

NAME	The name of the procedure function or package that has references to the REFERENCED_NAME. This view is not just for stored procedures. Views and other objects have dependencies as well.
TYPE	For PL/SQL program units, this includes PROCEDURE, FUNCTION, PACKAGE, and PACKAGE BODY.
REFERENCED_OWNER	This schema owns the referenced object. A procedure can insert a row into a STUDENTS table in another schema. The procedure is dependent on that STUDENTS table. This column is the owner of the STUDENTS table.
REFERENCED_NAME	This is the object name being referenced. This can be a table, view, sequence, or database link. This column will include all other procedures, functions, and packages being references, including packages in the data dictionary such as DBMS_OUTPUT.
REFERENCED_LINK_NAME	This is the name of a database link. A procedure could select from a table at a remote site with the syntax table_name@link_name. This would include the link name.
SCHEMAID	This is the schema ID of the referenced owner.
DEPENDENCY_TYPE	This is HARD unless the dependency is through a REF type object, then this column value is REF.

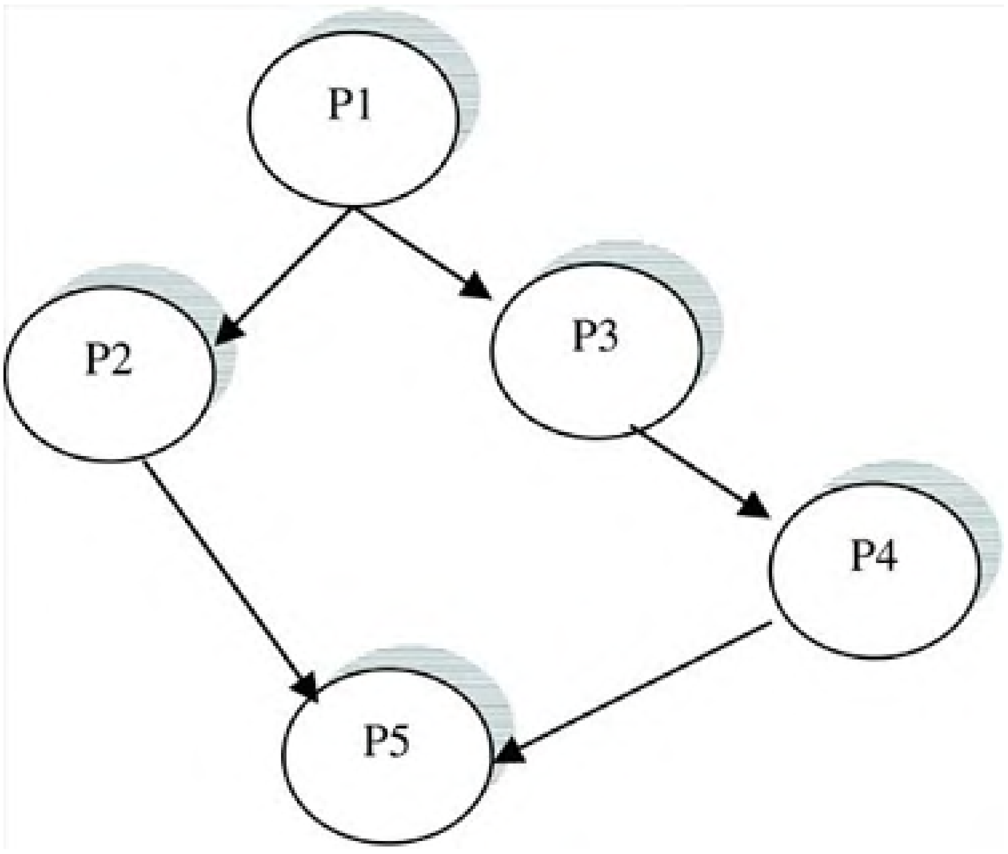
When a procedure is invalid, the task is to identify all objects that REFERENCE that procedure. To do this, SELECT NAME, TYPE WHERE REFERENCED_NAME = the invalid object. Continue this search until there are no more dependencies.

There are a variety of ways to automate this search. Writing a recursive PL/SQL function is one option. The following is one solution that populates a TEMP table with a SELECT from USER_DEPENDENCIES. To demonstrate a scenario, consider the impact to procedures P1 through P4, or any other procedures, when P5 becomes invalid. This dependency is illustrated in Figure 9-3. The procedure code for P1 through P5 is:

```
CREATE OR REPLACE procedure P5 IS
BEGIN
    dbms_output.put_line('executing p5');
END;
/
CREATE OR REPLACE procedure P4 IS
BEGIN
    P5;
END;
/
CREATE OR REPLACE procedure P3 IS
BEGIN
    P4;
END;
/
CREATE OR REPLACE procedure P2 IS
```

```
BEGIN
    P5;
END;
/
CREATE OR REPLACE procedure P1 IS
BEGIN
    P2;
    P3;
END;
/
```

Figure 9-3. Invalid Procedures.



Assume for the example that all procedures are invalid and we need to analyze what calls P5. What calls the code that calls P5? To trace a dependency, create a TEMP table with PARENT, CHILD columns.

```
CREATE TABLE temp (parent VARCHAR2(30), child VARCHAR2(30));
```

Load the starting point, which can be any point in the tree. We start with P5.

```
INSERT INTO temp VALUES ('P5', null);
```

Repeat the following INSERT statement until the message 0 rows created is received.

```
INSERT INTO temp
SELECT u.name a, t.parent b
FROM   user_dependencies u, temp t
WHERE  u.referenced_name=t.parent
AND NOT EXISTS
    (SELECT * FROM temp WHERE
      parent=u.name AND child=t.parent);
```

Select from TEMP, shown next, to show that P1 calls P2 and P3, P2 calls P5, and so forth.

```
PARENT              CHILD
-------------------- -------
P1                  P2
P1                  P3
P3                  P4
P2                  P5
P4                  P5
P5
```

Tracing invalid code should start with an attempt to compile the schema using DBMS_UTILITY, described in Section 9.4, "Dependencies among Procedures." After compiling the schema, trace invalid objects by identifying those objects that have an immediate dependency. For this example, identify procedures P4 and P2, because they directly call procedure P5.

Troubleshooting a dependency issue is only required when code is changed and recompiled without any preliminary analysis. There are numerous methods for tracing code dependency. The USER_DEPENDENCIES view provides all necessary information. You can automate a process with a PL/SQL solution or sequentially query the USER_DEPENDENCIES until you identify where the code is broken.

How can you extract the source code for P4 and P2? This may be necessary to investigate why the code is broken. Use the views USER and ALL_SOURCE covered in the next section.

9.6 USER_SOURCE

The source code for compiled procedures, functions, packages, or package bodies is accessible from the USER_SOURCE view.

Refer to Chapter 5 for a detailed explaination on the differences among data dictionary views that have prefixes: USER, ALL, and DBA. The PL/SQL code you compile in your schema can be selected at any time from USER_SOURCE.

- Query USER_SOURCE for the PL/SQL code you have compiled in you account.
- Query ALL_SOURCE for everything in USER_SOURCE plus all other PL/SQL to which you have been given the EXECUTE privilege.
- Query DBA_SOURCE is all the PL/SQL code in the database.

Because the scope of DBA views is everything in the database, you must have either the Oracle DBA role or the Oracle SELECT_CATALOG_ROLE role. The DBA role has high privileges. The SELECT_CATALOG_ROLE is intended for users who need to query data dictionary views. Application developers should be given this role.

The view USER_OBJECT uses the column names:

- OBJECT_NAME
- OBJECT_TYPE

The views USER_DEPENDENCIES and USER_SOURCE refer to the same components with a different name:

- NAME
- TYPE

You can always retrieve stored procedure code from this view. A procedure recreated and compiled with errors is retrievable from this view.

```
SQL> desc user_source
 Name                          Null?    Type
 ----------------------------- -------- ---------------
 NAME                                   VARCHAR2(30)
 TYPE                                   VARCHAR2(12)
 LINE                                   NUMBER
 TEXT                                   VARCHAR2(4000)
```

The columns in the USER_SOURCE view have the following description.

NAME	This is the name in the CREATE OR REPLACE clause. This is not the host file. Running the script @MY_HELLO.SQL with a CREATE OR REPLACE PROCEDURE HELLO statement creates the object name HELLO. The data dictionary stores all attributes in upper case.
TYPE	This is FUNCTION, PROCEDURE, PACKAGE, or PACKAGE BODY. There is never an underscore in PACKAGE BODY.
LINE	This identifies a line of text. A 50-line text file compiled as a stored procedure is 50 lines of text in the data dictionary. Oracle does not reformat the text. If the program fails at run time, a line number will be included in the error message. This should be matched with the LINE number in this view. It is always possible to identify the specific line within a stored procedure at which execution failed.
TEXT	This is a line of text as read from the original source file.

The text of a PL/SQL program is available through this view. You can query USER_SOURCE and redirect the code text to a SQL*Plus spool file. If the original source code is lost, you can redirect the text to a spool file with a SQL extension. The following SQL script queries the source for the HELLO procedure spooling it to a SQL file named MY_HELLO.SQL. The SQL*Plus command SET TERM suppresses output for the script while the source HELLO is spooled to the SQL file.

```
-- Filename generate_my_hello.sql
set feedback off
set pagesize 0
set term off
spool my_hello.sql
SELECT '-- Filename MY_HELLO.SQL' FROM dual;
SELECT 'CREATE OR REPLACE ' FROM dual;
SELECT text FROM user_source
WHERE name='HELLO';
SELECT '/' FROM DUAL;
spool off
set term on
set feedback on
set pagesize 1000
```

Running this script generates a file MY_HELLO.SQL. You can edit and recompile the HELLO procedure using this output file.

```
SQL> @generate_my_hello.sql
```

This produces the following text file, MY_HELLO.SQL, as shown next:

```
-- Filename MY_HELLO.SQL
CREATE OR REPLACE
PROCEDURE hello IS
BEGIN
   dbms_output.put_line('Hello');
END;
/
```

You select the specification of an Oracle package from ALL_SOURCE. The following generates a spool file DBMS_OUTPUT with the package specification and documentation for using this Oracle package.

```
set feedback off
set pagesize 0
set term off
spool dbms_output
SELECT text FROM all_source
WHERE name='DBMS_OUTPUT';
spool off
set term on
set feedback on
set pagesize 1000
```

The output from this script will be a text file, DBMS_OUTPUT.LST, shown next. This text, extracted from ALL_SOURCE, is the package specification as compiled in the database. The following illustrates the clarity of the interface definition and documentation that accompanies many of the Oracle packages.

```
package dbms_output as
  ------------
  --  OVERVIEW
  --
  --  These procedures accumulate information in a buffer
  --  (via "put" and "put_line") so that it can be retrieved
  --  out later (via "get_line" or "get_lines").  If this
  --  package is disabled then all calls to this package are
  --  simply ignored.  This way, these routines are only
  --  active when the client is one that is able to deal
  --  with the information.  This is good for debugging, or
  --  SP's that want to display messages or reports
  --  to sql*dba or plus (like 'describing procedures', etc.).
  --  The default buffer size is 20000 bytes.  The
  --  minimum is 2000 and the maximum is 1,000,000
  -----------
  --  EXAMPLE
  --
  --  A trigger might want to print out some debugging
  --  information.  To do this the trigger would do
  --  dbms_output.put_line('I got here:'||:new.col||' is
  --  the new value'); If the client had enabled the
  --  dbms_output package then this put_line would be
  --  buffered and the client could, after executing the
  --  statement (presumably some insert, delete or update
  --  that caused the trigger to fire) execute
  --   begin dbms_output.get_line(:buffer, :status); end;
  --  to get the line of information back.  It could then
  --  display the buffer on the screen. The client would
  --  repeat calls to get_line until status came back as
  --  non-zero.  For better performance, the client would
  --  use calls to get_lines which can return an array of
  --  lines.
  --
  --  SQL*DBA and SQL*PLUS, for instance, implement a
  --  'SET SERVEROUTPUT ON' command so that they know
  --  whether to make calls to get_line(s) after issuing
  --  insert, update, delete or anonymous PL/SQL calls
  --  (these are the only ones that can cause triggers or
  --  stored procedures to be executed).
  ------------
  --  SECURITY
  --
  --  At the end of this script, a public synonym
  --  (dbms_output) is created and execute permission on
  --  this package is granted to public.
  ----------------------------
  --  PROCEDURES AND FUNCTIONS
  --
  procedure enable (buffer_size in integer default 20000);
  pragma restrict_references(enable,WNDS,RNDS);
  --  Enable calls to put, put_line, new_line, get_line
  --    and get_lines. Calls to these procedures are
  --    noops if the package has not been enabled.
  --    Set default amount of information to buffer.
  --    Cleanup data buffered from any dead sessions.
  --    Multiple calls to enable are allowed.
  --  Input parameters:
  --    buffer_size
  --      Amount of information, in bytes, to buffer.
  --      Varchar2, number and date items are stored in
  --      their internal representation.  The information
  --      is stored in the SGA. An error is raised if the
  --      buffer size is exceeded.  If there are multiple
  --      calls to enable, then the buffer_size is generally
  --      the largest of the values specified, and will
  --      always be >= than the smallest value
  --      specified. Currently a more accurate determination
  --      is not possible.  The maximum size is 1,000,000,
  --      the minimum is 2000.
  procedure disable;
  pragma restrict_references(disable,WNDS,RNDS);
  --  Disable calls to put, put_line, new_line, get_line
  --    and get_lines. Also purge the buffer of any remaining
  --    information.
  procedure put(a varchar2);
  pragma restrict_references(put,WNDS,RNDS);
  procedure put(a number);
  pragma restrict_references(put,WNDS,RNDS);
  --  Put a piece of information in the buffer.
  --    When retrieved by get_line(s), the number and
  --    date items will be formated with to_char using
  --    the default formats. If you want another format
  --    then format it explicitly.
  --  Input parameters:
  --    a
  --      Item to buffer
  procedure put_line(a varchar2);
  pragma restrict_references(put_line,WNDS,RNDS);
  procedure put_line(a number);
  pragma restrict_references(put_line,WNDS,RNDS);
  --  Put a piece of information in the buffer followed by
  --    an end-of-line marker.  When retrieved by get_line(s),
  --    the number and date items will be formated with
  --    to_char using the default formats.  If you
  --    want another format then format it explicitly.
  --    get_line(s) return "lines" as delimited by "newlines".
  --    So every call to put_line or new_line will generate a
  --    line that will be returned by get_line(s).
  --  Input parameters:
  --    a
```

```
    --      Item to buffer
    --  Errors raised:
    --    -20000, ORU-10027: buffer overflow, limit of
    --    <buf_limit> bytes.
    --    -20000, ORU-10028: line length overflow, limit
    --    of 255 bytes per line.
    procedure new_line;
    pragma restrict_references(new_line,WNDS,RNDS);
    --  Put an end-of-line marker.  get_line(s) return "lines"
    --    as delimited by "newlines".  So every call to
    --    put_line or new_line will generate a line that will
    --    be returned by get_line(s).
    --  Errors raised:
    --    -20000, ORU-10027: buffer overflow, limit of
    --    <buf_limit> bytes.
    --    -20000, ORU-10028: line length overflow, limit
    --    of 255 bytes per line.
    procedure get_line(line out varchar2, status out integer);
    pragma restrict_references(get_line,WNDS,RNDS);
    --  Get a single line back that has been buffered.
    --    The lines are delimited by calls to put_line or
    --    new_line.  The line will be constructed taking all
    --    the items up to a newline, converting all the items
    --    to varchar2, and concatenating them into a single
    --    line. If the client fails to retrieve all lines before
    --    the next put, put_line or new_line, the non-retrieved
    --    lines will be discarded. This is so if the client is
    --    interrupted while selecting back the information,
    --    there will not be junk left over which would
    --    look like it was part of the NEXT set of lines.
    --  Output parameters:
    --    line
    --      This line will hold the line - it may be up to 255
    --      bytes long.
    --    status
    --      This will be 0 upon successful completion of the
    --      call. 1 means that there are no more lines.
    type chararr is table of varchar2(255) index
      by binary_integer;
    procedure get_lines(lines out chararr,
      numlines in out integer);
    pragma restrict_references(get_lines,WNDS,RNDS);
    --  Get multiple lines back that have been buffered.
    --    The lines are delimited by calls to put_line or
    --    new_line.  The line will be
    --    constructed taking all the items up to a newline,
    --    converting all the items to varchar2, and
    --    concatenating them into a single line. Once get_lines
    --    is executed, the client should continue to retrieve
    --    all lines because the next put, put_line or new_line
    --    will first purge the buffer of leftover data. This is
    --    so if the client is interrupted while selecting back
    --    the information, there will not be junk left over.
    --  Input parameters:
    --    numlines
    --      This is the maximum number of lines that the
    --      caller is prepared to accept. This procedure will
    --      not return more than this number of lines.
    --  Output parameters:
    --    lines
    --      This array will hold the lines - they may
    --      be up to 255 bytes long each.  The array is indexed
    --      beginning with 0 and increases sequentially. From a
    --      3GL host program the array begins with whatever is
    --      the convention for that language.
    --    numlines
    --      This will be the number of lines actually returned.
    --      If it is less than the value passed in, then there
    --      are no more lines.
  end;
```

9.7 Sharing Code

You create an Oracle account with a username, password, and basic roles that allow that user to create tables and procedures. You add a new user named BLAKE to the database with the following:

```
CREATE USER BLAKE IDENTIFIED BY BLAKE
DEFAULT TABLESPACE STUDENT_DATA
TEMPORARY TABLESPACE TEMP;
GRANT CONNECT, RESOURCE TO BLAKE;
```

From this point forward, BLAKE can create tables and stored procedures. Another user SCOTT is also creating procedures. By default, BLAKE cannot see SCOTT'S objects and SCOTT cannot see BLAKE'S objects.

Sharing of objects is done on a per-object bases. BLAKE can grant table and package access to SCOTT on an as-needed basis. BLAKE has a package that provides selected payroll information. The PAYROLL package queries data from a sensitive SALARIES table. The SALARIES table is restricted. However, BLAKE can selectively give access to the PAYROLL package. BLAKE controls the type of salary information available through the procedures and functions he defines in the PAYROLL package. To grant SCOTT the right to use the PAYROLL package, BLAKE executes the following:

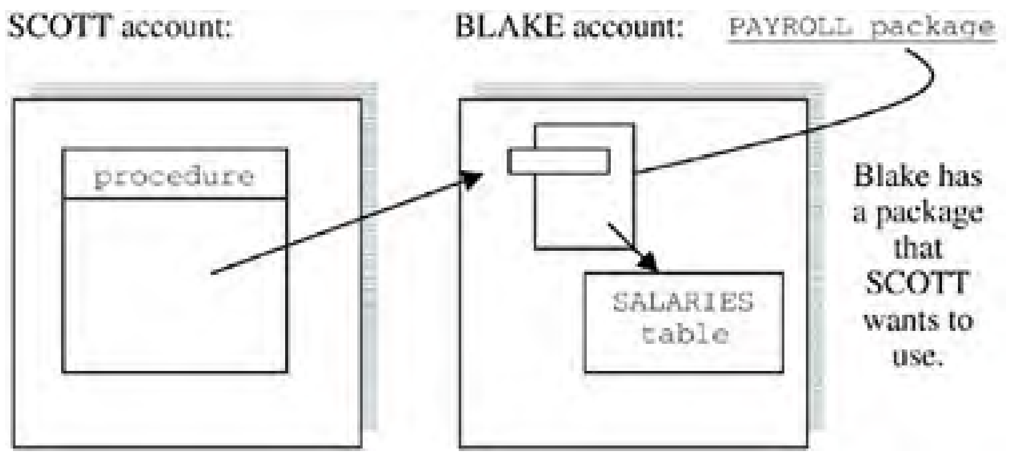
```
SQL> GRANT EXECUTE ON PAYROLL TO SCOTT;
```

SCOTT can now execute any procedure or function defined in the PAYROLL package. SCOTT creates a private synonym:

```
SQL> CREATE SYNONYM PAYROLL FOR BLAKE.PAYROLL;
```

From this point forward, SCOTT can write PL/SQL stored procedures that use the PAYROLL package. SCOTT cannot access the SALARIES table or any other object. Figure 9-4 illustrates this scenario.

Figure 9-4. Sharing Code.



BLAKE can revoke the GRANT EXECUTE with the following:

```
SQL> REVOKE EXECUTE ON PAYROLL FROM SCOTT;
```

This immediately invalidates the procedure in the SCOTT account. If SCOTT runs the PL/SQL procedure that uses the PAYROLL package, it will fail. If BLAKE reactivates the grant, then Oracle will resolve the invalid procedure. It will recognize that SCOTT'S procedure is valid and run it.

BLAKE can revoke the privilege from SCOTT; he can grant the privilege any time. Each time it will be resolved. There should be some communication between BLAKE and SCOTT so that SCOTT can recompile the PL/SQL code when this occurs—this alleviates Oracle from having to resolve an invalid object at run time.

Ideally, if SCOTT'S code becomes invalid, he can execute an ALTER/COMPILE command or use DBMS_UTILITY to validate the code. This topic is covered in Section 9.4, "Dependencies among Procedures."

If BLAKE wants to share his code, he must GRANT EXECUTE to a user. BLAKE cannot grant execute privileges to SCOTT through a role. When Oracle compiles or revalidates SCOTT'S procedures, it looks at the privileges in the SCOTT account. It does not look at roles. The success or failure of compiling SCOTT'S code depends on SCOTT'S privileges; roles are ignored. The following is a scenario that explains this concept.

Consider the following:

HR_ADMIN is a senior role. HR is a less privileged role. The following creates the two roles. Because HR_ADMIN is a senior role, all lesser roles are granted to HR_ADMIN.

```
SQL> CREATE ROLE HR_ADMIN;
SQL> CREATE ROLE HR;
SQL> GRANT HR TO HR_ADMIN;
```

The following SQL creates the user SCOTT and grants five roles to SCOTT. However, the default roles for SCOTT are CONNECT and RESOURCE. When SCOTT connects to the database, he only has the CONNECT and RESOURCE roles. SCOTT can enable the other roles with a SET ROLE command.

```
SQL> CREATE USER SCOTT IDENTIFIED BY TIGER;

SQL> GRANT CONNECT, RESOURCE, HR, HR_ADMIN TO SCOTT;

SQL> ALTER USER SCOTT DEFAULT ROLE CONNECT, RESOURCE;
```

The following scenario supposes that roles are used when compiling procedures—this is not the case. This scenario is to illustrate why. Suppose BLAKE grants execute on PAYROLL to HR_ADMIN. SCOTT connects to the database with default roles CONNECT, and RESOURCE. When SCOTT connects to the database, the procedure that uses the PAYROLL package is invalid. This is because the HR_ADMIN is not a default role and has not been set. SCOTT executes the command to enable the role.

```
SQL> SET ROLE HR_ADMIN;
```

SCOTT has now enabled the role to which the package execute privilege was given. This scenario can be more complicated. Application programs set roles, based on the username, to enable various parts of an application. Roles can be granted to other roles. The dynamics of roles allows run-time execution of the following:

```
SQL> SET ROLE HR_ADMIN, HR;
SQL> SET ROLE HR;
```

Given the dynamics of roles and their use within applications, it is reasonable that user accounts be the basis for procedure grants, as shown in Figure 9-4.

You cannot do the following:

- `CREATE ROLE HR;`
- `GRANT EXECUTE ON PAYROLL TO HR;`
- `GRANT HR TO SCOTT;`
- Expect SCOTT to write a PL/SQL procedure that uses BLAKE'S package.

BLAKE must

```
GRANT EXECUTE ON PAYROLL TO SCOTT;
```

9.8 Compilation Dependency

Concerning just packages, a specification must exist and compile without errors before the package body can compile.

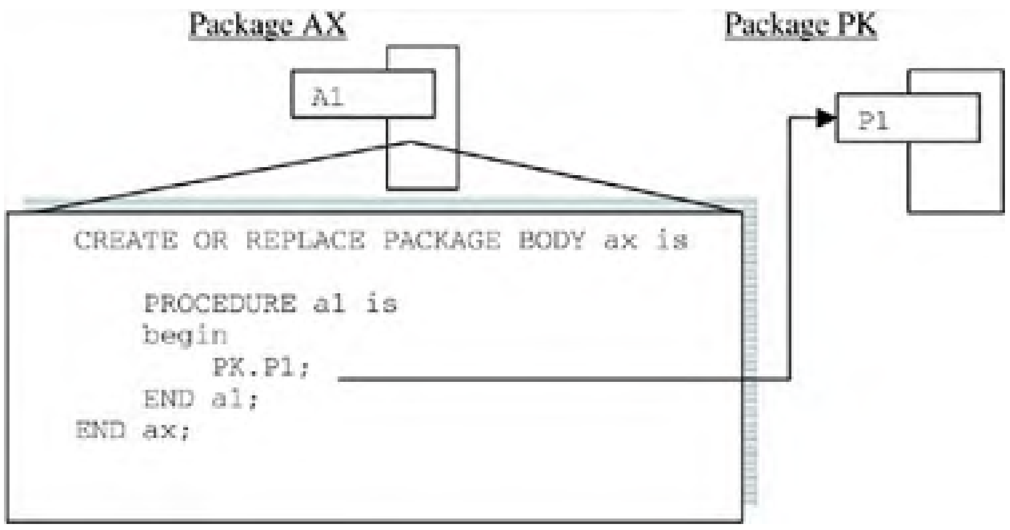
9.8.1 Scenario 1

The following is a package specification and body. There is one procedure.

```
-- Filename PK_SPECIFICATION.SQL
CREATE OR REPLACE PACKAGE pk is
    PROCEDURE p1;
END pk;
/

-- Filename PK_BODY.SQL
CREATE OR REPLACE PACKAGE BODY pk is
    PROCEDURE p1 IS
    begin
        dbms_output.put_line('execute p1');
    END p1;
END pk;
/
```

Figure 9-5 illustrates the package PK and the visible procedure P1. Additionally, there is a second package, AX. An AX procedure uses the procedure P1.

Figure 9-5. Package Body Referencing Another Package.



```
CREATE OR REPLACE PACKAGE BODY ax is

    PROCEDURE a1 is
    begin
        PK.P1;
    END a1;
END ax;
```

The body of AX is dependent on the specification of PK. Specifically, the body of the procedure A1 calls PK.P1.

The body of AK becomes invalid if we recompile the PK specification. The body of AK becomes invalid if we recompile the AK specification—recompiling a package specification always invalidates the body.

Recompiling the PK body or AX body has no impact to other code. This is always the case. For any package, recompiling the package body is an isolated event that will not affect any other stored procedure.

Recompiling a package specification will invalidate the associated package body. It may also affect the bodies of other packages. This would be the case when the body code calls procedures in the recompiled spec.

The following summarizes the package compilation order.

- Recompiling PK specification forces:
    - Recompiling PK body and AK body.

- Recompiling AK specification forces:
    - Recompiling AK body.
- Recompiling either PK body or AK body affects nothing.

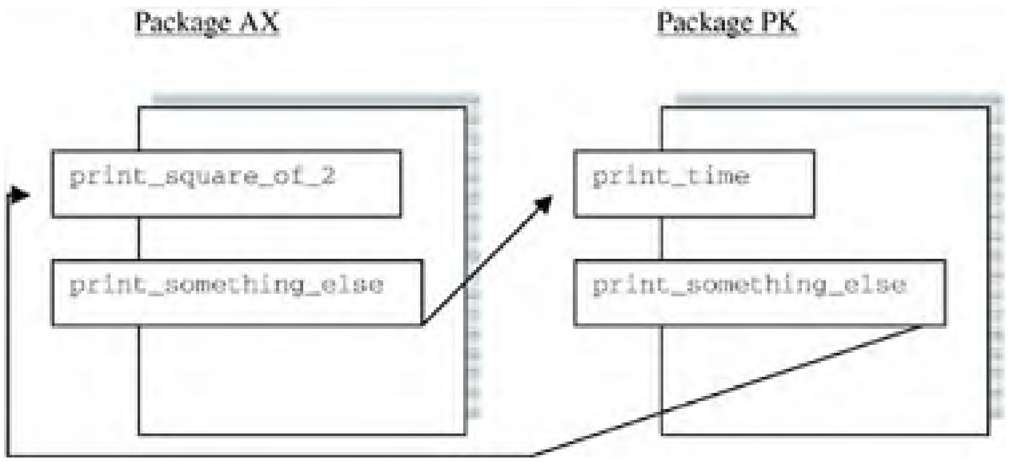
9.8.2 Scenario 2

Two packages can have their body dependent on each other's package specification. The following illustrates this case.

The first package is PK. The second package is AX. A PK procedure references an AK procedure. An AX procedure references a PK procedure. See Figure 9-6.

- PK.print_something_else calls AX.print_square_of_2
- AX.print_something_else calls PK.print_time

Figure 9-6. Two Interdependent Packages.



To compile the four program units, two specs and two bodies, the package specifications must be compiled first, then the bodies. The code for this shows the package specifications first, then the bodies.

```
CREATE OR REPLACE PACKAGE pk is
    PROCEDURE print_time;
    PROCEDURE print_something_else;
END pk;
/

CREATE OR REPLACE PACKAGE ax is
    PROCEDURE print_square_of_2;
    PROCEDURE print_something_else;
END ax;
/

CREATE OR REPLACE PACKAGE BODY pk is
    PROCEDURE print_time IS
    begin
        dbms_output.put_line(SYSDATE);
    END print_time;
    PROCEDURE print_something_else IS
    BEGIN
        ax.print_square_of_2;
    END print_something_else;
END pk;
/

CREATE OR REPLACE PACKAGE BODY ax is
    PROCEDURE print_square_of_2 IS
    begin
```

```
        dbms_output.put_line(SQRT(2));
    END print_square_of_2;

    PROCEDURE print_something_else IS
    BEGIN
        pk.print_time;
    END print_something_else;
END ax;
/
```

Recompiling has the following effects.

- Recompiling PK specification forces:
    - Recompiling PK body and AK body.
- Recompiling AK specification forces:
    - Recompiling AK body and PK body.
- Recompiling either PK body or AK body affects nothing.

Recompiling a body affects nothing, but recompiling a specification could have wide implications by invalidating the body of that package plus many other package bodies.

9.9 USER_ERRORS

SHOW ERRORS is a SQL*Plus command. It returns information from the USER_ERRORS view, which has the following description:

```
SQL> desc user_errors
 Name                            Null?    Type
 ------------------------------- -------- ----------------
 NAME                            NOT NULL VARCHAR2(30)
 TYPE                                     VARCHAR2(12)
 SEQUENCE                        NOT NULL NUMBER
 LINE                            NOT NULL NUMBER
 POSITION                        NOT NULL NUMBER
 TEXT                            NOT NULL VARCHAR2(4000)
```

You can select compiler error results when not running SQL*Plus by querying this view. The USER_ERRORS view contains errors for objects compiled in your schema. ALL_ERRORS and DBA_ERRORS are other views with wider scope. Refer to Chapter 5 for a complete description of the differences between the USER, ALL, and DBA data dictionary views. The following is a description of the columns.

NAME	This is the name in the CREATE OR REPLACE clause. This is not the host file. Running the script @MY_HELLO.SQL with a CREATE OR REPLACE PROCEDURE HELLO statement creates the object name HELLO. The data dictionary stores all attributes in upper case.
TYPE	This is FUNCTION, PROCEDURE, PACKAGE, or PACKAGE BODY. There is never an underscore in PACKAGE BODY.
SEQUENCE	This corresponds to the error number relative to the number of errors in the compile.
LINE	This is the list-file line number, which contains the error. This corresponds to the LINE column in USER_SOURCE.
POSITION	This is the column position of the error.
TEXT	This contains the text of the error. For example: `PLS-00302: component 'PUTLINE' must be declared`

Chapter Ten. PL/SQL Program Units

This chapter presents PL/SQL from a software engineering perspective. PL/SQL has the following program units:

- Procedure
- Function
- Package specification
- Package body

Stored procedures are PL/SQL program units that exist as objects in the Oracle database. The term "stored procedure" refers to a compiled and callable program within the database. Oracle implements stored procedures as procedures, functions, and packages. You do not call packages. You call the procedures and functions within the visible part of the package. Database vendors have different implementations for stored procedures. SQL Server Transact-SQL provides procedures and functions but not packages.

Subprograms refer to procedures and functions in general, whether it be a stand-alone procedure/function or procedure/function in a package. PL/SQL procedures and functions model procedures and functions of other languages—they receive parameters, declare local variables, and implement some degree of logic using the constructs of the PL/SQL programming language. Packages are program units that encapsulate persistent data as well as subprograms.

PL/SQL is a loosely typed language. It handles implicit conversion; for example, you can pass a NUMBER type argument to a procedure designed to accept a string. On the other hand, Oracle performs type checking at compile-time and run-time. A compile-time constraint check fails if a procedures attempts to assign a composite record structure to a scalar variable. Run-time errors occur when an implicit conversion fails. Variables of type NUMBER are implicitly converted to a VARCHAR2; however, if the run-time value is 9999 and the range is VARCHAR2(3), a run-time constraint violation occurs because 9999 is too large for the three-byte character string.

All subprogram parameters have a mode that designates the direction of the data—between the caller and called program. Parameter modes are IN, OUT, and IN OUT. Most violations of this type are detected at compile-time; for example, a compile error occurs when an IN mode variable is written to. This is detected when the IN mode variable is on the left side of an assignment statement. In general, there are many forms of constraint and type checking performed by the compiler. Run-time checks enforce constraints not evaluated at compile-time.

PL/SQL is a language that integrates several software engineering principals: information hiding, encapsulation, data abstraction, modular design, and stepwise refinement. A key feature of the language is the package paradigm that couples the language with design. A compiled package specification can be reviewed as a detail design document, minimizing the transition from design to program development.

Additionally, PL/SQL integrates the ANSI SQL Standard into the programming language in a way that facilitates rapid development. Examples of these features are SQL SELECT statements that populate composite record structures and cursor FOR loops.

Structure type definitions can be created in the database from which database table column types and PL/SQL array structures can be derived. The integration of types between the database and PL/SQL allows a program to select a column that stores an array of structures in a PL/SQL variable.

The robust features of this language provide a strong motivation for Java and .Net developers to implement large portions of their functionality with PL/SQL.

A strength that continues to grow with PL/SQL is Oracle support for additional built-in packages. The list of packages provided by Oracle is quite extensive. Following are some examples of listings.

- DBMS_PIPS and DBMS_ALERT provide a mechanism by which separate Oracle processes can communicate asynchronously—similar to UNIX-named pipes.
- Advanced Queuing includes several packages that push and pull data into and out of logical queues in a distributed environment.
- The DBMS_LOB package provides support for handling large 4-gigabyte objects as column values in a table. This allows a PL/SQL procedure to load a large document, such as a Microsoft Word or PDF file, in native format, into a table.
- Java in the database allows a PL/SQL package specification to act as a PL/SQL API to a body implementation using Java. An example would be a PL/SQL package specification written in PL/SQL with a package body written in Java.
- PL/SQL programs can perform IO to host files using the UTL_FILE package.
- There are packages for generating random numbers; implementing HTML server-side programs, similar to a CGI model or an XML DOM API; and more. This list is relatively short compared to the complete sweep of APIs provided by Oracle.

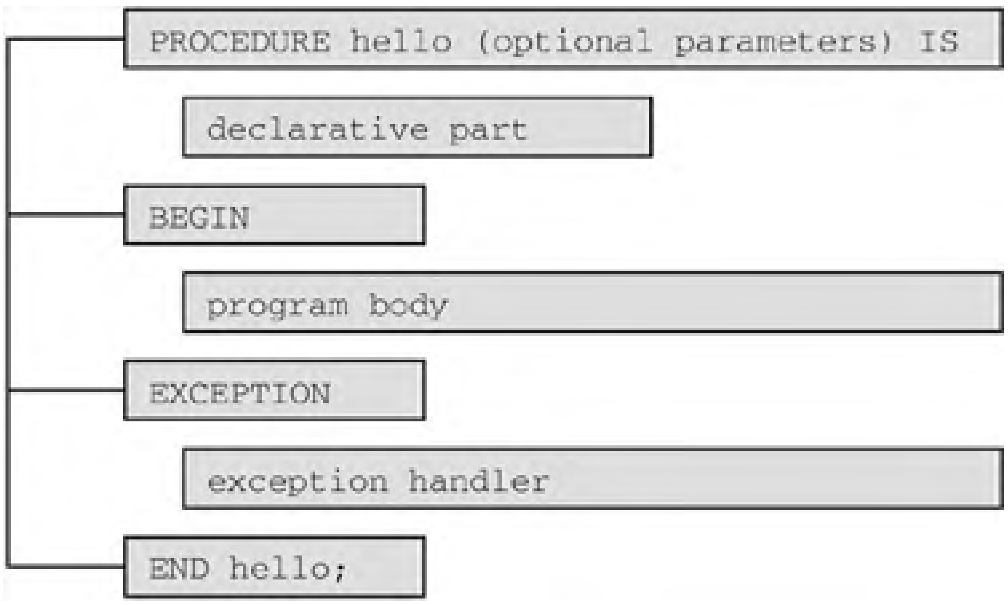
You can execute a stored procedure from a SQL*Plus session. You can embed PL/SQL code in third-generation languages: C programs written with Pro*C can include PL/SQL blocks and call stored procedures. Any client-side interface, whether Web or client/server, can invoke PL/SQL procedures compiled in the database. Connectivity support for PL/SQL procedures includes ODBC, JDBC, and Oracle's Net8.

There is benefit to PL/SQL as a language choice for Oracle systems development. Compared to other languages, it takes less time to code a complex function written in PL/SQL. The following discussion addresses some considerations when planning what packages you might develop and how they might interface with each other.

10.1 Procedures

A PL/SQL procedure is a stand-alone program that you compile into an Oracle database schema. Procedures can accept arguments. When you compile a procedure, the procedure identifier of the CREATE PROCEDURE statement becomes the object name in the data dictionary. Figure 10-1 illustrates the components and keywords of a procedure. This structure applies for a stand-alone procedure and the procedure code of a package body.

Figure 10-1. Procedure.



The key components of a procedure to which you write code are the following:

Declarative Part	This is where you declare variables, for example: `local_counter NUMBER := 0;` You may also have type definitions. Type definitions would be required for composite structures such as records and index-by tables. You may code procedures and functions within the declarative part. For example, you might write a local function that returns a substring with certain characteristics. You could use this local function to make the body easier to read. You can declare exceptions here. If you declare an exception within a procedure you should handle it locally. No program that calls your procedure can capture exceptions that are local to your procedure.
Subprogram Body	The subprogram body contains the logical algorithm implemented with PL/SQL control constructs. PL/SQL supports loops, if-then-else structures, case statements, and declare-block structures.
Exception handler	The exception handler is optional. The exception handler is similar to the try-catch model in other languages. You can code an exception handler for a specific type of error or write a general-purpose exception handler.

You should name a procedure with a verb. Procedures usually perform some action such as update the database, write to a file, or send a message.

A procedure need not have parameters. When creating a procedure do not use parentheses if there are no parameters. Close parentheses are optional when calling the procedure. The following illustrates the absence of parentheses.

First, create a table.

```
CREATE TABLE TEMP(n NUMBER);
```

The procedure INSERT_TEMP inserts a row into the table TEMP.

```
PROCEDURE insert_temp IS
BEGIN
    INSERT INTO TEMP (n) VALUES (0);
END insert_temp:
```

A procedure that uses INSERT_TEMP can code either of the following statements:

```
insert_temp;
insert_temp();
```

You can code IS or AS—either syntax is acceptable.

```
PROCEDURE insert_temp IS | AS
```

Appending the procedure name to the END clause is optional, but highly recommended. A procedure can span several screens. When scrolling through code, it is extremely helpful to see an END clause and know you have not skipped reading into the next package procedure.

```
END; -- acceptable.

END procedure_name; -- highly recommended in production.
```

A common procedure style is to align the IS, BEGIN, EXCEPTION, and END. Indent all code within these keywords. This style makes the code easier to read. The following illustrates a procedure that prints the average and sum of values in a table. The code section between the IS and BEGIN is called the declarative part of the procedure.

```
PROCEDURE print_temp
IS
    v_average NUMBER;
    v_sum     NUMBER;
BEGIN
    SELECT AVG(n), SUM(n) INTO v_average, v_sum
    FROM TEMP;

    dbms_output.put_line('Average:'||v_average);
    dbms_output.put_line('Sum:'||v_sum);
END print_temp;
```

A stand-alone procedure frequently evolves into a new package or merges with an existing package. Consider the procedure, INSERT_TEMP, shown on p. 251, which inserts a number into the TEMP table. The migration of INSERT_TEMP into a package is a simple editing process that produces the following:

```
PACKAGE temp_operations IS
    PROCEDURE insert_temp;
END temp_operations;

PACKAGE BODY temp_operations IS
    PROCEDURE insert_temp IS
    BEGIN
        INSERT INTO temp (n) VALUES (0);
    END insert_temp;
END temp_operations;
```

The user of INSERT_TEMP changes their PL/SQL to use the new interface:

```
temp_operations.insert_temp;
temp_operations.insert_temp();
```

10.2 Functions

Some languages have a single subprogram type. A Java class has methods and a method may or may not return data. The C language is the same. The only subprogram in C is a function. A C function can return a value or modify the contents of an address value passed as an argument.

PL/SQL includes two types of subprograms: procedures and functions. Consider a package to be the encapsulation of operations on some object. That object can be a table, database pipe, host file, tables in a remote database, or many other options. Procedures take actions on the object and modify them. Functions provide a means to acquire status or state information about the object.

The package typically behaves as an API that hides an object and provides operations on the object. If the user needs to know, for example, how large the object is, a method must be available. Functions play this role. Functions frequently act as selectors on an object.

Consider the function to be the evaluation of an attribute of an object. Functions are not actors, but rather evaluations of a state. You should use nouns to name functions.

Figure 10-2 illustrates the components of a function. The key components of a function, to which you write code, are the following:

Declarative Part

This is where you declare variables, for example:

```
local_counter NUMBER := 0;
```

You may also have type definitions. Type definitions would be required for composite structures such as records and index-by tables. You may code procedures and functions within the declarative part. For example, you might write a local function that returns a substring with certain characteristics. You could use this local function to make the body easier to read.

You can declare exceptions here. If you declare an exception within a procedure you should handle it locally. No program that calls your procedure can capture exceptions that are local to your procedure.

You frequently declare a variable that is the variable to be returned. You must return something in a function. If the function returns a NUMBER, the NUMBER variable could be declared. That variable would be used in the RETURN statement.

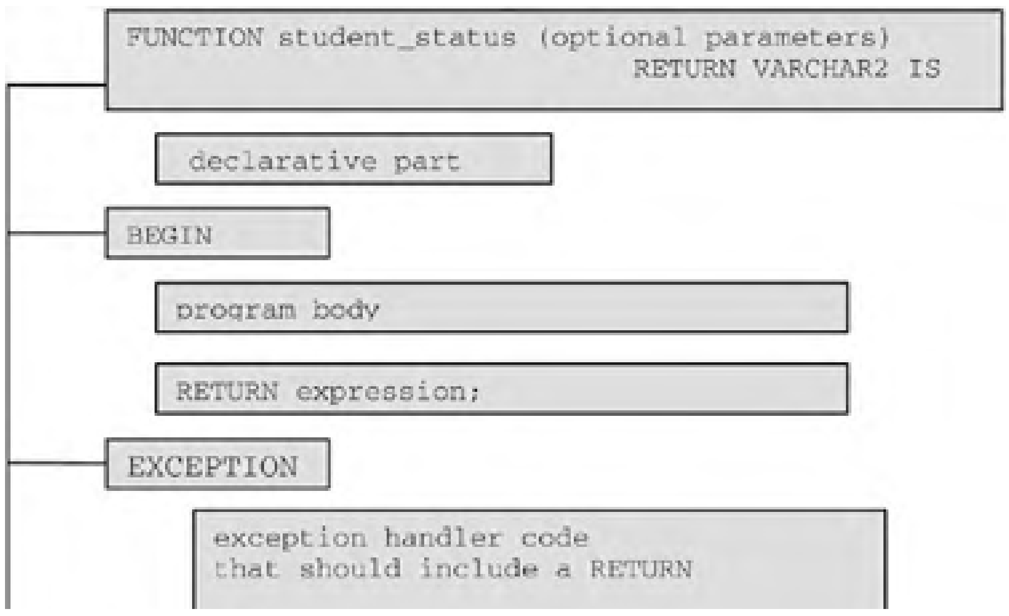
Subprogram Body

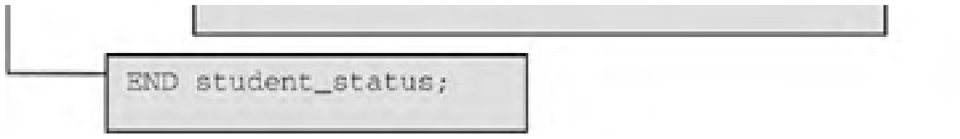
The subprogram body contains the logical algorithm implemented with PL/SQL control constructs. PL/SQL supports loops, if-then-else structures, case statements, and declare-block structures. The body must include a RETURN statement, otherwise it will not compile.

Exception Handler

The exception handler is optional. The exception handler is similar to the try-catch model in other languages. You can code an exception handler for a specific type of error or write a general-purpose exception handler. Make sure the exception handler includes a RETURN statement.

Figure 10-2. Function.





Parameters are optional, but the RETURN part is not. The FUNCTION statement must include a RETURN and a type.

The RETURN statement, in the subprogram body, is a key component. A function compilation fails if there is no RETURN statement in the body of the code. A function that follows an execution path that ends before executing a RETURN statement causes a run-time error. A function must have a RETURN statement to compile; it must execute a RETURN during run-time. A frequent oversight is to write an exception handler and not include a RETURN clause. A function that enters an exception handler without a RETURN statement generates a run-time error.

The following example is a function that returns a DATE type. The declarative part creates the variable NEXT_DAY part. The program body assigns a value and returns that value.

```
CREATE OR REPLACE FUNCTION tomorrow RETURN DATE
IS
    next_day DATE;
BEGIN
    next_day := SYSDATE + 1;
    RETURN next_day;
END tomorrow;
```

Functions can include an expression in the RETURN statement. This often means less code in the function body. The following function performs the same logic as the preceding function. In this example, the local variable is not necessary.

```
FUNCTION tomorrow RETURN DATE IS
BEGIN
    RETURN SYSDATE + 1;
END tomorrow;
```

If there are no parameters, do not use empty parentheses in the function definition. This rule applies to procedures as well. The user of the function can use the function as a single expression that evaluates to some type. The following code uses the TOMORROW function in an assignment statement. The same procedure uses the function as an argument to DBMS_OUTPUT.

```
CREATE OR REPLACE PROCEDURE tomorrow_user
IS
    my_date DATE;
BEGIN
    my_date := tomorrow();
    my_date := tomorrow;
    dbms_output.put_line('Tomorrow is '||tomorrow);
END tomorrow_user;
```

Use functions to increase the readability of the code. The following uses the TOMORROW function in a loop. The code is clear and identifies when the loop stops.

```
WHILE local_date_variable < tomorrow LOOP
    Loop body
END LOOP;
```

Variables and function calls are always interchangeable provided the function returns the same type as the variable. For example, the TOMORROW function returns a DATE type. Any PL/SQL statement, in which a DATE variable is acceptable, can use the function TOMORROW.

```
PROCEDURE sample
IS
    today DATE;
BEGIN
    today := tomorrow - 1;
    dbms_output.put_line(tomorrow - 1);
END sample;
```

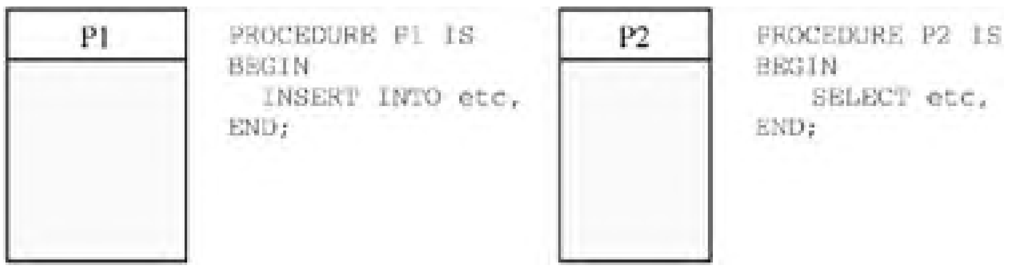
10.3 Subprogram Encapsulation: Introduction to Packages

Packages provide a mechanism to logically group smaller program units together. The grouping of procedures into a package modularizes the code. Physically, the use of packages means fewer files and fewer modules to manage. Programmers are more inclined to reuse modular code.

Programmers who begin working on the system, built with packages, quickly grasp the overall architecture because they can learn the system in a top-down fashion—looking at whole modules and how they interface.

Figure 10-3 illustrates two procedures. To grant the use of these procedures to other applications requires a GRANT EXECUTE on each procedure. Imagine 50 procedures. Additional functionality means more procedures, more grants, and more files.

Figure 10-3. Stand-Alone Procedures as Components in an Application.



```
PROCEDURE P1 IS
BEGIN
   INSERT INTO etc.
END;
```

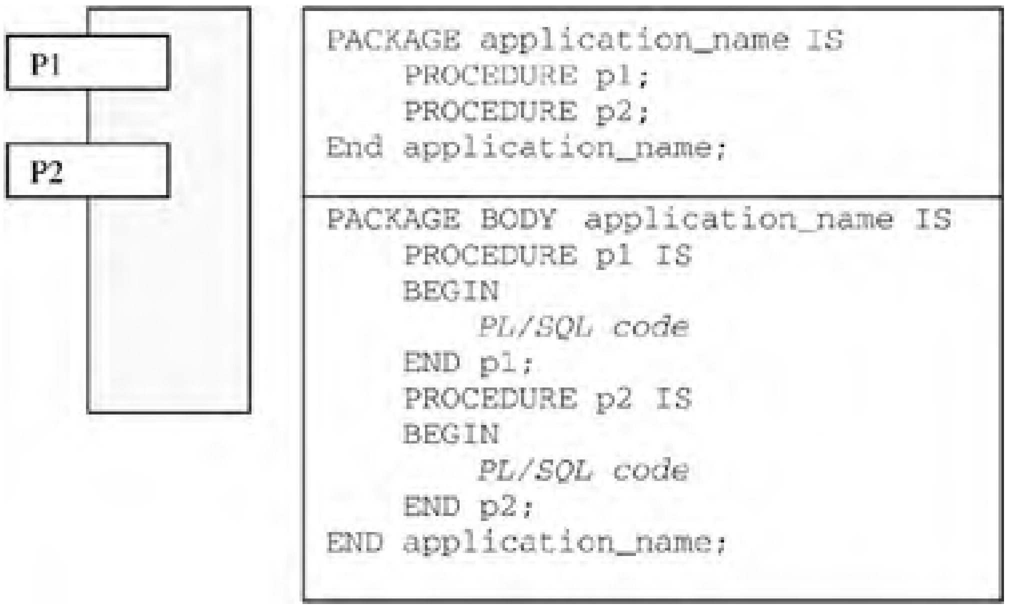
```
PROCEDURE P2 IS
BEGIN
    SELECT etc.
END;
```

Additionally, a stand-alone procedure can send and receive scalar parameters with a type of NUMBER, VARCHAR2, and DATE. The language allows you to define composite structures such as records. A stand-alone procedure cannot return a composite record structure unless another package specification defines that structure. Relying on individual procedures and functions is limiting in large-scale development.

Procedures P1 and P2 can be easily merged into a single package. This makes sense if these procedures perform related operations. Procedure P1 does an INSERT. Procedure P2 does a SELECT. Consider these SQL statements to be on the same table.

The migration process includes copying the procedure body of each into a single package body. The procedure interface definitions become the package specification. After this, you include new procedures and functions to enhance the overall functionality of the package. Figure 10-4 shows the encapsulation of procedures P1 and P2 into a single package.

Figure 10-4. Encapsulation of Procedures.



```
PACKAGE application_name IS
    PROCEDURE p1;
    PROCEDURE p2;
End application_name;

PACKAGE BODY application_name IS
    PROCEDURE p1 IS
    BEGIN
        PL/SQL code
    END p1;
    PROCEDURE p2 IS
    BEGIN
        PL/SQL code
    END p2;
END application_name;
```

After the package is complete, the specification (shown in the top right of Figure 10-4) is compiled, then the body. Additional procedures and functions are added to the package.

10.4 Package Specification

Programming has always involved the "packaging" of related programs. This includes compiling a group of programs into a single object library and configuring the code of a single subsystem under one source code directory. Configuration management tools allow us to package our software under logical and functional subject areas so developers can easily locate programs for check-out and check-in purposes.

For some environments, the highest abstraction of a unit of software is a procedure. The management of the software repository then becomes the management of many individual programs. For other languages, procedures that collectively satisfy a major functional requirement compile into one object module, managed as one software unit. This is the case with C programming where multiple C functions compile as one C object file. Java packages contain Java classes and the compiler relies on a match between directory and package name.

A programmer who needs to use a procedure must first identify the module and then identify the specific function, procedure, or method needed. The C language uses a header file, a separate file from the C algorithmic code, that defines the interface—this is the first step in learning how to use another programmer's C code.

Complexity of a programming language and the programming environment hinders development. IDEs are very successful with helping developers quickly locate reusable programs that are part of the development environment. Many of these IDE tools allow you to add your newly developed code to the library repository—this enables a developer to search for a particular interface and easily locate the API of another developer. Oracle's JDeveloper and Procedure Builder are IDE tools that provide this type of rapid development environment support.

Without the use of an IDE, the features of a language can lead to small amounts of code reuse. One of the most powerful features of PL/SQL is the simplicity of the package specification. The PL/SQL language requires that the interface to a program collection compile into a single program unit. This program unit, the package specification, is not complex. It simply defines the API. The package body contains the details of the logic.

You can incorporate PL/SQL package specification in the design process. High-level design includes defining high-level interfaces and documenting what an implementation will do. The package specification identifies interfaces and includes and uses code and comments to define the interface. A set of package specifications can represent the complete top-level design of a system. A package body with pseudocode and comments can represent the detailed design.

In theory, should you see the code to a package specification on a programmer's desk, you should not know whether they are at the tail end of the design phase or the beginning of the code phase. The package specification overlaps the design and coding phase. The package specification defines a set of interfaces and operations performed on a set of objects, such as tables in a schema. It is also a compiled object in the database, subject to syntax rules and database privileges.

The package specification is a single ASCII file that compiles as a single program unit. The package body is also a single ASCII file. The body compiles only after a successful compilation of the specification. You can put the specification in the same file as the body.

10.4.1 Syntax and Style

The basic package specification syntax is:

```
CREATE PACKAGE package_name IS
   Type definitions for records, index-by tables,
      varrays, nested tables
   Constants
   Exceptions
   Global variable declarations
   PROCEDURE procedure_name_1 (parameters & types);
   PROCEDURE procedure_name_2 (parameter & types);
   FUNCTION function_name_1 (parameters & types) RETURN type;
END package_name;
```

There is no order to the procedures and functions in the specification. Put them in a logical order that suits the developer.

The package body will include the PL/SQL code for each of the subprograms in the specification. Each subprogram in the specification must have an accompanying subprogram body.

The package specification can declare data types such as a record type, data declarations such as a record, and exceptions. All data objects declared in the package specification are global. Therefore, only declare what needs to be global.

The PROCEDURE statement in the package body, including the subprogram name, parameter names, parameter modes, and parameter types, must match the PROCEDURE statement in the specification. The same holds true for the FUNCTION subprogram.

The package body template for the preceding specification is:

```
CREATE PACKAGE BODY package_name IS
   PROCEDURE procedure_name_1 (parameters & types)
   IS
      local variables
   BEGIN
      body of code
   END procedure_name_1;
   PROCEDURE procedure_name_2 (parameter & types)
   IS
      local variables
   BEGIN
      body_of_code
   END procedure_name_2;
   FUNCTION function_name_1 (parameters & types) RETURN type
   IS
      local variables
   BEGIN
      body of code
      RETURN statement;
   END function_name_1;
END package_name;
```

A package specification may declare exceptions. Let's modify PACKAGE_NAME so it declares an exception. The exception name is INVALID_OPERATION. The package code is the same, but with an exception declaration. Exceptions are sometimes declared all in the beginning of the spec, sometimes all at the bottom. Most important, comments should indicate which subprograms potentially raise which exceptions.

```
CREATE PACKAGE package_name IS
   invalid_operation EXCEPTION;
   PROCEDURE procedure_name_1 (parameters & types);
   PROCEDURE procedure_name_2 (parameter & types);
   FUNCTION function_name_1 (parameters & types) RETURN type;
END package_name;
```

The user of PACKAGE_NAME has a question: "What procedures raise this exception?" Assuming the answer is PROCEDURE_NAME_1, the application code, should it choose to handle a possible exception, looks like this:

```
BEGIN
   other code, etc
   package_name.procedure_name_1(parameters);
   other code, etc
EXCEPTION
   WHEN package_name.invalid_operation THEN do something;
END;
```

A package specification can also declare type definitions such as records. If a procedure is to return a student name and status, the option exists to declare a record type in the specification, as follows:

```
CREATE PACKAGE package_name IS

   TYPE student_rec_type IS RECORD (
      student_name students.student_name%TYPE,
      status students.status%TYPE);

   PROCEDURE get_student
      (stud_id  IN students.student_id%TYPE
       stud_rec OUT student_rec_type);

END package_name;
```

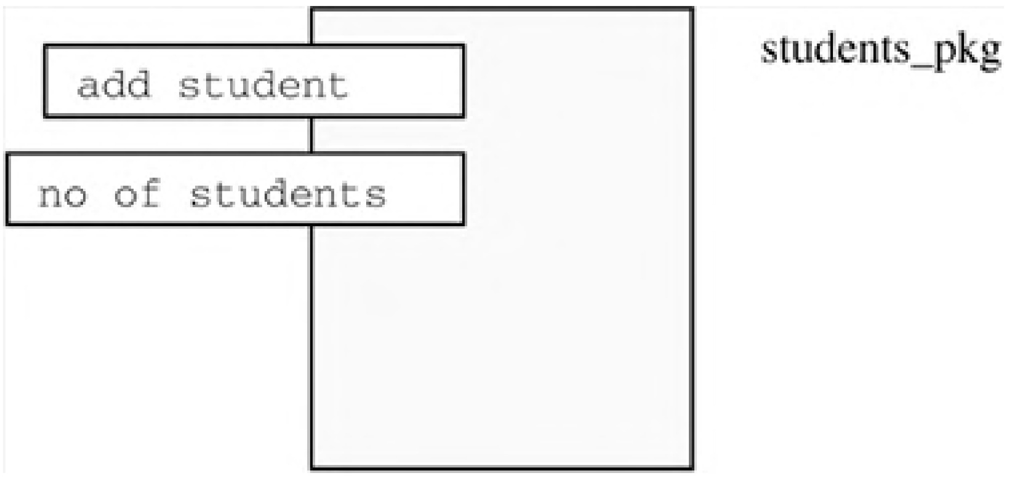
The package users should drive the design. You should consider the user community when designing the specification. Overloading of procedures and functions is a frequent technique to making a package interface acceptable to a wide audience. A reasonable suggestion to the prior PACKAGE_NAME specification is to declare a function in the spec and have this return a student record type. Why? Because it can make the application using the package easier to write and easier to read. Such a modification changes the specification to the following:

```
CREATE PACKAGE package_name IS

    TYPE student_rec_type IS RECORD (
        student_name students.student_name%TYPE,
        status students.status%TYPE);

    PROCEDURE get_student
        (stud_id  IN students.student_id%TYPE
         stud_rec OUT student_rec_type);

    FUNCTION student
        (stud_id  IN students.student_id%TYPE)
    RETURN student_rec_type;
END package_name;
```

10.4.2 Developing a Specification

A Booch (Grady Booch) diagram, showing just one package in Figure 10-5, is an excellent paradigm for design. These diagrams are easy to whiteboard for technical group discussion. They provide a clear image of the software architecture. Booch diagrams specify what operations exist. This enables one to look at the system requirements and judge if the operations designed will satisfy the requirements.

Figure 10-5. Students Package.



We begin this section with a software requirement to implement procedures that perform operations on the STUDENTS table. Chapter 4 includes the DDL, entity diagram, and sample data for a student application.

We have some software to write. First, we need to implement the following:

- Add a new student.
- Return the total number of students.

We begin with a model depicting these operations. These operations belong together in a package because they each perform functions on the student's table. The package name will be STUDENTS_PKG.

The development must transition from the graphical model to an interface design. We use the package specification as a vehicle for describing this interface. The following paragraphs illustrate the development toward an interface specification.

ADD_STUDENT SUBPROGRAM

Below is a list of the columns of the STUDENTS table. Also refer to the STUDENTS table DDL in the data model demo of Chapter 4 (p. 146).

- STUDENT_ID
- STUDENT_NAME

- COLLEGE_MAJOR
- STATUS
- STATE
- LICENSE_NO

The STUDENT_ID column is the primary key and is generated with the sequence STUDENTS_PK_SEQ. This column value is determined in the body of the ADD_STUDENT procedure and will use the sequence attribute NEXTVAL during the INSERT statement.

The three columns after STUDENT_ID are mandatory column values. The last two are optional. We should make STATE and LICENSE_NO optional parameters in the procedure call. These will have a default of NULL in the procedure interface.

```
PROCEDURE add_student
(v_student_name   IN  students.student_name%TYPE,
 v_college_major  IN  students.college_major%TYPE,
 v_status         IN  students.status%TYPE,
 v_state          IN  students.state%TYPE DEFAULT NULL,
 v_license_no     IN  students.license_no%TYPE DEFAULT NULL);
```

An application program can use this interface with the following call:

```
students_pkg.add_student(name, major, status);
```

The interface also permits:

```
students_pkg.add_student(name,major,status,state,license_no);
```

NO_OF_STUDENTS SUBPROGRAM

This should be a function. The requirement is to return an attribute of the entire student body. The following is a function with some flexibility. The user can use this function to get the number of students who have a specific major, a specific status, or a combination. The two parameters passed would be (1) the subject major description value from the MAJOR_LOOKUP table and (2) a status value of "Degree" or "Certificate."

```
FUNCTION NO_OF_STUDENTS
(v_major  IN major_lookup.major_desc%TYPE DEFAULT NULL,
 v_status IN students.status%TYPE DEFAULT NULL)
RETURN NUMBER;
```

This interface permits the package user to write the following calls:

```
-- get the number of students with an undeclared major.
undeclared_major_count INTEGER :=
   students_pkg.no_of_students(v_major=> 'Undeclared ');

-- get the total number of students.
student_count INTEGER := students_pkg.no_of_students();

-- get the number of degree-seeking biology students.
biology_degrees INTEGER :=
   students_pkg.no_of_students
        (v_major => 'Biology',
         v_status => 'Degree');
```

The STATUS column indicates the student's status with the school and has a CHECK constraint with valid values of "Degree" and "Certificate."

PACKAGE SPECIFICATION

The package specification now defines the interfaces. It initially appears to meet the requirements of adding a student and returning a count of students in the school. This is a starting point. We can add subprograms as needed. Additional procedures and functions can enhance the overall functionality of the package.

```
CREATE OR REPLACE PACKAGE students_pkg IS
  PROCEDURE add_student
    (v_student_name   IN  students.student_name%TYPE,
     v_college_major  IN  students.college_major%TYPE,
     v_status         IN  students.status%TYPE,
     v_state          IN  students.state%TYPE DEFAULT NULL,
     v_license_no     IN  students.license_no%TYPE DEFAULT NULL);

  FUNCTION NO_OF_STUDENTS
    (v_major  IN major_lookup.major_desc%TYPE DEFAULT NULL,
     v_status IN students.status%TYPE DEFAULT NULL)
  RETURN NUMBER;
END students_pkg;
```

10.5 Package Body

The following is the body for the student's package. It includes just procedure code for the insert and a select for a student count.

This code could be enhanced with error handling logic; for example, it should include exception-handling logic for the case where the V_STATUS parameter violates the CHECK constraint (refer to pp. 103–104 for CHECK constraint exception handling.)

```
CREATE OR REPLACE PACKAGE BODY students_pkg IS
  PROCEDURE add_student
    (v_student_name   IN  students.student_name%TYPE,
     v_college_major  IN  students.college_major%TYPE,
     v_status         IN  students.status%TYPE,
     v_state          IN  students.state%TYPE DEFAULT NULL,
     v_license_no     IN  students.license_no%TYPE DEFAULT NULL)
  IS
  BEGIN
      INSERT INTO students VALUES
        ('A'||students_pk_seq.NEXTVAL,
          v_student_name,
          v_college_major,
          v_status,
          v_state,
          v_license_no);
  END add_student;

  FUNCTION NO_OF_STUDENTS
    (v_major  IN major_lookup.major_desc%TYPE DEFAULT NULL,
     v_status IN students.status%TYPE DEFAULT NULL)
  RETURN NUMBER
  IS
    ccount INTEGER;
  BEGIN
      SELECT COUNT (*) INTO ccount
      FROM   students, major_lookup
      WHERE  students.college_major = major_lookup.major
      AND    major_lookup.major_desc =
                 nvl(v_major,major_lookup.major_desc)
      AND   students.status = nvl(v_status,students.status);
    RETURN ccount;
  END NO_OF_STUDENTS;
END students_pkg;
```

The development of the body can lead to other local procedures and functions. These are hidden, also called private. Consider the structure of the student's package body, which is shown next. The notation in the package body, *declarative part*, identifies the region were we can optionally declare variables global to all procedures in the body. In this region we can declare types, exceptions, and procedures—everything in this region is local to the package body, yet global to all procedures declared beneath it.

```
PACKAGE BODY students_pkg IS

   Declarative part.

   PROCEDURE add_student( etc, )
   END;
   FUNCTION no_of_student( etc, )
   END;
END students_pkg;
```

There are two options when adding local subprograms to the declarative part. One is to code the subprogram bodies sequentially; however, there can be no forward referencing among the subprogram bodies. Consider two local procedures. The body looks like this.

```
PACKAGE BODY students_pkg IS
   PROCEDURE local_1 ( etc, ) IS
   BEGIN
       body code
```

```
  END;
  PROCEDURE local_2 ( etc, ) IS
  BEGIN
     body code
  END;
  PROCEDURE add_student( etc, ) IS
     body code
  END;
  FUNCTION no_of_student( etc, ) RETURN etc IS
     body code
  END;
END students_pkg;
```

In this package body, ADD_STUDENT and NO_OF_STUDENTS can reference LOCAL_1 and LOCAL_2. The procedure LOCAL_2 can reference LOCAL_1 but LOCAL_1 cannot reference LOCAL_2—the compiler, at the time it compiles LOCAL_1, has no known declaration of LOCAL_2.

The style of the preceding package is most common. Local procedures frequently exist at the top of the package. There is usually not much interdependency among local procedures in a body. An alternative is to first declare the definition of local procedures (called an early declaration), then the body. This code would look like the following:

```
PACKAGE BODY students_pkg IS

  PROCEDURE local_1 ( etc, );
  PROCEDURE local_2 ( etc, );

  PROCEDURE local_1 ( etc, ) IS
  BEGIN
     body code
  END;
  PROCEDURE local_2 ( etc, ) IS
  BEGIN
     body code
  END;
  PROCEDURE add_student( etc, ) IS
     body code
  END;
  FUNCTION no_of_student( etc, ) RETURN etc IS
     body code
  END;
END students_pkg;
```

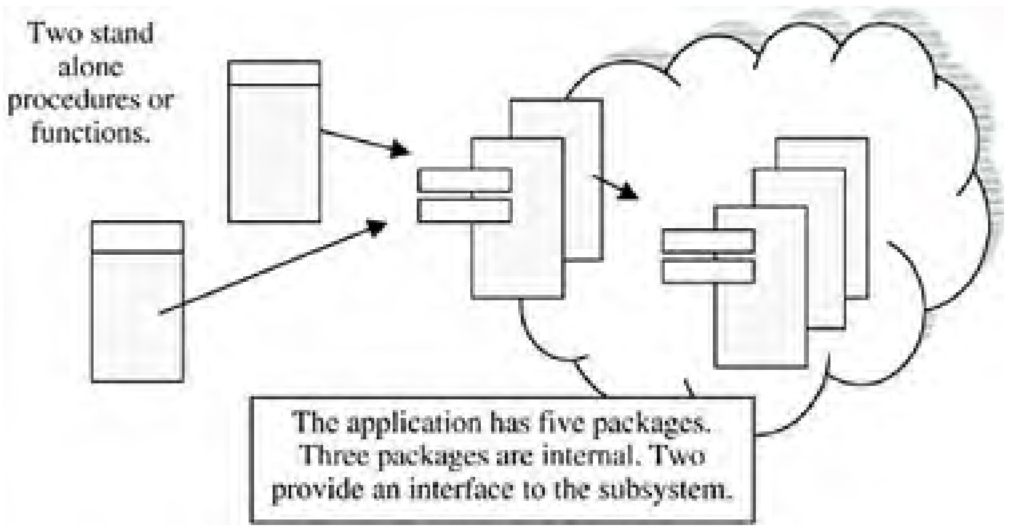
Because all subprogram definitions occur prior to subprogram bodies, any subprogram can call any other subprogram. Furthermore, the local subprogram bodies can be placed in any order. For this package body, LOCAL_1 and LOCAL_2 subprogram bodies can be placed last in the package body.

[Team LiB] PREVIOUS NEXT

10.6 Application Partitioning

Figure 10-6 illustrates a subsystem, also called an application, that has visible and private package components. The contrast between the white and shaded part denotes a package specification part as distinct from a package body. The white, "exposed" slots represent components of an API. The API defines the interface and, in general, specifies what the package does. The body is the implementation. To see the implementation (i.e., how the code performs its task), one must look at the package code.

Figure 10-6. Booch Diagram: Subsystem with Packages and Stand-Alone Programs.


Two stand alone procedures or functions.
The application has five packages. Three packages are internal. Two provide an interface to the subsystem.

The design of a system begins with partitioning functionality into subcomponents, or subsystems. The process continues with partitioning the subsystem into packages. A single package within a subsystem satisfies a set of related requirements. A subsystem will likely include Oracle built-in packages (e.g., UTL_FILE for File IO). A package can logically reside in multiple subsystems, also called applications.

We can implement the concept of a subsystem using the existing database framework, specifically the database schema. To make a subsystem, we can encapsulate packages as objects within a schema and use database grants to make some packages visible and some not. The packages, for which we gave grants, become the API of the subsystem. A programmer might develop a useful and general purpose dates package that would exist in numerous subsystems.

Figure 10-7 illustrates a subsystem for a Student Registration Application. In contains a set of packages that support all the functionality for student registration. One package is exposed. This is the API. It provides ADD and GET operations on the private data stored in the tables of the schema and supports other application code that requires these services. For example, a student housing application might require the GET service component of the API.

Figure 10-7. Student Information System Consisting of Several Applications.

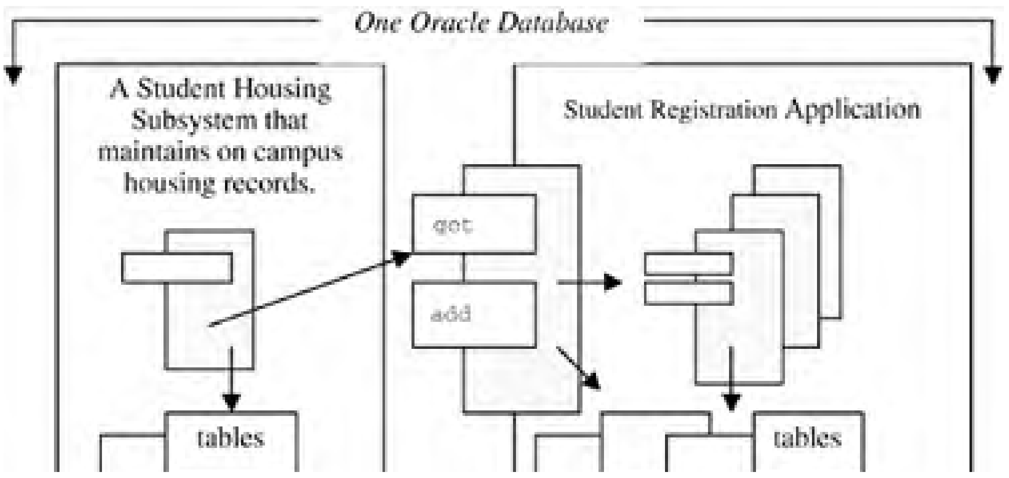

One Oracle Database
A Student Housing Subsystem that maintains on campus housing records.
Student Registration Application
get
add
tables
tables

Figure 10-7 is drawn to illustrate how applications within a single database can be partitioned and how the logic within partitions can interconnect through an API. This diagram shows a single database. Application can easily be distributed among several databases and achieve the same level of interconnectivity using database links.

The Student Registration Application (see Figure 10-7) shows three packages that are not part of the visible API. These are not callable from other applications. The application, as drawn literally, consists of four packages and four tables. One package is visible.

We grant execute privileges to users who need the API. Once the application becomes production, we can further lock-down the schema and revoke CREATE TABLE and similar roles from the registration production schema. From a database perspective, this includes the following:

- Create a database account for the student registration schema. We call this Student Registration Application (SRA).
- Build the application. Create all tables and packages in the SRA schema.
- GRANT EXECUTE ON API packages to other users such as the Student Housing Application owner.
- Revoke the Oracle CONNECT and RESOURCE roles from the SRA. This prevents accidental updates to production by development and test teams that work on separate development and test database instances.
- There is now a restricted one-way access to the SRA system. This is the package or set of packages that form the API.
- The application will undergo enhancements and new releases. This will require enabling the application roles, CONNECT and RESOURCE, and other roles to permit database table and PL/SQL enhancements.

10.7 Data Abstraction

You can simplify complex application logic by building abstract structures on top of the existing language structures. The following package implements a STACK built using a PL/SQL index-by table. This is a last-in-first-out (LIFO) stack of student names. This demonstrates the ability to bind data and operations together and restrict the interface to the definitions in the package specification.

The user of this package must conform to the push/pop operations in the specification. Although the package body implements this structure with an index-by table, the implementation could be a database table, a set of tables, or tables in another database—this uses a PL/SQL table.

The stack package has the following operations and exceptions:

Subprogram	Purpose
CLEAR	This procedure empties the stack.
PUSH	This pushes a name onto the stack.
POP	This is overloaded. You can use the function to pop a name. You can also use the procedure form.
STACK_NOT_EMPTY	This returns TRUE if the stack has any names.
Exceptions	
UNDERFLOW	A pop of an empty stack is an exception, similar to subtracting 1 from a POSITIVE number that has a current value of 1. A divide by zero is an exception because it produces a result not in the set of numbers. We raise the exception UNDERFLOW following the pop of an empty stack.

First, look at how one might use the stack. We want an interface to be simple. The following PL/SQL code selects student names from the STUDENTS table and pushes the names onto the stack. Following that the names are poped within a call to DBMS_OUTPUT.

```
BEGIN
    FOR rec IN (SELECT student_name FROM students) LOOP
        students_stack.push(rec.student_name);
    END LOOP;

    WHILE (students_stack.stack_not_empty) LOOP
        dbms_output.put_line(students_stack.pop);
    END LOOP;
END;
```

Running the PL/SQL block produces the following names:

```
William
Steven
Kathryn
Mary
John
```

The specification and body for the stack are shown next. You can easily modify this for a FIFO LIST structure. The component stored in the index-by table is a VARCHAR2. You can build a stack of STUDENT%ROWTYPE structures.

```
CREATE OR REPLACE PACKAGE students_stack IS
    --
    -- Empties the stack.
    --
    PROCEDURE clear_stack;
    --
    -- Push a name onto the stack.
    --
    PROCEDURE push (name IN students.student_name%TYPE);
    --
    -- Pop and return a name from the stack.
    --
    PROCEDURE pop (name OUT students.student_name%TYPE);
    FUNCTION pop RETURN students.student_name%TYPE;
```

```
  --
  -- Check status. Return true if stack has data
  --
  FUNCTION stack_not_empty RETURN BOOLEAN;

  -- This exception is raised with a pop
  -- on an empty stack.
  --
  UNDERFLOW exception;

END students_stack;
```

The package body for this stack encapsulates the stack that is implemented as a PL/SQL index-by table.

```
CREATE OR REPLACE PACKAGE BODY students_stack IS
  TYPE table_type IS TABLE OF students.student_name%TYPE
  INDEX BY BINARY_INTEGER;

  the_stack table_type;

  stack_pointer BINARY_INTEGER := 0;

  PROCEDURE clear_stack IS
  BEGIN
    stack_pointer := 0;
  END clear_stack;

  PROCEDURE push (name IN students.student_name%TYPE) IS
  BEGIN
    stack_pointer := stack_pointer +1;
    the_stack(stack_pointer) := name;
  END push;

  PROCEDURE pop (name OUT students.student_name%TYPE) IS
  BEGIN
    name := the_stack(stack_pointer);
    stack_pointer := stack_pointer—1;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN RAISE underflow;
  END pop;

  FUNCTION pop RETURN students.student_name%TYPE IS
  BEGIN
    stack_pointer := stack_pointer—1;
    RETURN the_stack(stack_pointer + 1);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      stack_pointer := stack_pointer + 1;
      RAISE underflow;
  END pop;

  FUNCTION stack_not_empty RETURN BOOLEAN IS
  BEGIN
    RETURN (stack_pointer > 0);
  END stack_not_empty;
END students_stack;
```

10.8 Parameters and Modes

Procedures are subprograms that generally perform some service—they act on or change something. A function generally just returns information. All subprogram parameters have a mode that is syntactically coded between the variable and the type definition. PL/SQL programs have three modes:

- IN (default)
- IN OUT
- OUT

IN	An IN mode parameter is passed to a subprogram with the understanding that the subprogram uses the parameter as a constant, read-only value. A parameter passed with IN mode can be a literal expression, constant declaration, or variable declaration. In the case of a variable, this mode provides some measure of security on the part of the calling program. The caller knows that upon completion of the call, the variable is unchanged. The compiler enforces this security. There are two checks. First, if the compiler detects an IN mode parameter to the left of an assignment, the compile fails. Second, if the variable is passed to a subsequent procedure that uses it as an OUT or IN OUT parameter, the compile fails.
IN OUT	Parameters passed in this mode can only be variables, not literals or constants. The assumption is that the called procedure will change the content passed. The called procedure will read as well as write to the structure content. An example is a procedure that passes a string to a data-scrubbing procedure. The scrubbing procedure runs the string through a loop checking for invalid characters and removes unwanted bytes. Upon completion of the scrubbing, the procedure that made the call has a clean string.
OUT	Parameters passed in this mode can only be variables, not literals or constants. Within a subprogram, the initial value of an OUT mode parameter is NULL. The intent of using an OUT mode parameter is to convey something about the interface—the caller has no obligation to pass data. The called procedure can write to the data structure; it can read the structure. For example, a sort procedure might be passed as an OUT mode structure. The process of sorting requires an iterative process of read/write operations. The called procedure only reads the structure to perform the sort. Any OUT mode parameter can change to IN OUT without any impact. However, this may not be consistent with the original design. If parameters arbitrarily change from OUT to IN OUT, this misrepresents the program interface and confuses the programmer who wonders, "What data is supposed to be passed?"

10.8.1 IN Mode (Default) is a Constant

An IN mode parameter is a constant or must be treated as a constant. A procedure that accepts an IN mode argument is restricted from writing to that argument. The following procedure will not compile because line 3 writes to the IN mode variable.

```
1  PROCEDURE print_next_value(v_data IN INTEGER) IS
2  BEGIN
3      v_data := v_data+1;  -- compile error
4      dbms_output.put_line(v_data);
5  END;
```

The following procedure, which performs the same service, does compile. The constant can be in an expression. This is the correct use of an IN mode parameter.

```
1  PROCEDURE print_next_value(v_data IN INTEGER) IS
2  BEGIN
3      dbms_output.put_line(v_data+1);
4  END;
```

In the following PL/SQL block, all calls to the previous PRINT_NEXT_VALUE are valid. Line 7, in the PL/SQL block, passes a variable. Following the call to PRINT_NEXT_VALUE, the block retains the original value of MY_DATA. The author of the print procedure declares the mode as IN, which informs users, "I will not change your data."

```
1  DECLARE
2      zero CONSTANT INTEGER := 0;
3      my_data INTEGER := 2;
4  BEGIN
5      print_next_value( 123 );
6      print_next_value( zero );
7      print_next_value( my_data );
8  END;
```

10.8.2 IN OUT Mode

There is an assumption drawn from looking at a procedure that takes an IN OUT mode parameter. That assumption is that the caller must provide data. This would be the IN part of an IN OUT argument. The following procedure, on line 4, reads from and writes to the data passed. This parameter must be IN OUT. It truly modifies the data passed.

```
1  PROCEDURE change_data(v_data IN OUT INTEGER) IS
2  BEGIN
3      for i in 1..10 loop
4          v_data := v_data + 1;
5      end loop;
6  END;
```

The caller of CHANGE_DATA knows that the before and after values of the IN OUT parameter may differ.

- MY_DATA before the call is 0.
- The value of MY_DATA after the block execution is 10.

```
1  DECLARE
2      my_data INTEGER := 0;
3  BEGIN
4      print_data(my_data);
5      dbms_output.put_line('block print: '||my_data);
6  END;
```

If this PL/SQL block declares MY_DATA as a constant, block execution fails.

10.8.3 OUT Mode

OUT mode parameters convey critical information about the interface. The intent is that the caller has no obligation to provide content to the caller. An OUT mode variable, upon entry to the procedure, is NULL.

In the following procedure, prior to the execution of line 4, V_DATA is NULL. A value of 100 is assigned. During execution, the procedure reads and writes to the parameter. From the beginning of program logic, PROVIDE_DATA makes no assumption about the content of V_DATA.

```
1  PROCEDURE provide_data(v_data OUT INTEGER)
2  IS
3  BEGIN
4      v_data := 100;
5      FOR i IN 1..10 LOOP
6          v_data := v_data +1;
7      END LOOP;
8  END;
```

A user of PROVIDE_DATA is the following PL/SQL block. The first INSERT into TEMP will be the value 0. The second insert will be the value 110.

```
1  DECLARE
2      my_data INTEGER := 0;
3  BEGIN
4      insert into temp values (my_data);
5      provide_data(my_data);
6      insert into temp values (my_data);
7  END;
```

10.8.4 Functions and Modes

The default mode for procedures and functions is IN mode. A general practice, to make code clearer, is to always type the mode, even when using the default. The convention with a function is that they are selectors—they return a specific piece of information. They also perform a processing-type service where data is passed in, changed, and returned with an IN OUT mode argument. Examples are built-in string functions: REPLACE and TRANSLATE.

Functions are often named using a noun; procedures are named with a verb. The large percentage of functions in any application will have all arguments passed with the IN mode, but functions can be declared with parameters that have all three modes. Functions with parameters other than IN mode should be rare.

The use of OUT mode arguments with a function provides a technique for some simple interfaces, especially when the function truly is a selector but must provide dual pieces of information. A function design can require that it fetch a record, but also returns the success of that fetch. Certainly a procedure can accomplish this, but it may be desirable, from an API perspective, to provide a function that follows this design.

The decision of function or procedure should, foremost, consider the readability and simplicity of the code that will use this interface. The following illustrates a function design that returns data plus a status. For this interface, assume ARG_1 is a primary key value used to identify the precise record to fetch. The argument NEXT_REC is the wanted data.

```
FUNCTION next_rec(arg1 IN type, next_record OUT type)
RETURN BOOLEAN;
```

This design allows the user to write code as follows:

```
WHILE (next_rec(arg1, my_record_structure))
LOOP
    process my_record_structure;
END LOOP;
```

An alternative to the function NEXT_REC is a procedure. The procedure solution would be the following—we rename the subprogram with a verb.

```
PROCEDURE get_next_rec(arg1 IN type,
                       next_record OUT type,
                       status OUT BOOLEAN);
```

From a user's perspective, we can use this procedure as well—a slightly different interface. The code block still uses a loop to fetch all records.

```
LOOP
    get_next_rec(arg1, my_record_structure, status)
    EXIT WHEN NOT status;
    process my_record_structure;
END LOOP;
```

From the PL/SQL block perspective, there is little difference. In concept, the function is a selector that evaluates to the next record, but optionally provides status about that fetch. The procedure behaves as a service, fetching a record, and returning the record and a status.

10.8.5 Named versus Positional Notation

Consider a procedure with the following interface definition.

```
PROCEDURE proc_name (arg1 mode and type, arg2 mode and type);
```

A user has two syntax options. The first is POSITIONAL notation, the second is NAMED notation:

```
1. proc_name(variable_1, variable_2);
2. proc_name(arg1 => variable_1, arg2 => variable_2);
```

What does the term Formal Parameter Name refer to? The formal parameter name refers to the name used in the interface definition of the procedure or function. For the preceding procedure, PROC_NAME, the formal parameter names are ARG1 and ARG2.

Formal parameter names should be generic and still convey what the parameter is used for. The following procedure definition uses formal names that convey to the user the intent of each parameter—the formal parameter names are FILE_ID and RECORD_READ:

```
PROCEDURE get_record (file_id     IN  INTEGER,
                      record_read OUT VARCHAR2);
```

One can look at this procedure definition and be fairly certain how to use this code. Additional information may be included in the procedure or package documentation. The user of this procedure has two calling styles. The first is positional notation.

```
DECLARE
    file_id             INTEGER;
    next_payroll_record VARCHAR2(100);
BEGIN

    get_record(file_id, next_payroll_record);

END;
```

Is this PL/SQL block clear—as far as what the code does? The PL/SQL block uses variable names that convey their use—this makes it fairly clear what is happening in this code. The PL/SQL block can use named notation, shown next:

```
DECLARE
    file_id             INTEGER;
    next_payroll_record VARCHAR2(100);
BEGIN

    get_record
        (file_id => file_id,
         record_read => next_payroll_record);

END;
```

Is this block any clearer? The call to GET_RECORD is correct but the information is redundant. Because of well-chosen variable names, formal parameter notation, in this case, is not necessary and, to most readers, makes the code wordy and more difficult to read. When do you use named notation? There are several cases when named notation is useful.

1. Use named notation when the variable names you choose do not completely convey their use. For example, you may pass a literal value and use named notation as a code documentation tool.
2. Use named notation when the subprogram is coded with default values and you are only using some of the defaults.

The first condition, passing a literal in a procedure, can occur with IN mode parameters. It may not be clear what the literal is used for. This is more likely to occur when your code interfaces with packages that are outside the present schema. Examples are other applications and Oracle packages. The following is a PL/SQL block that submits a job to the Oracle DBMS_JOB package. This block will dispatch a request to the job queue. The request is to run, asynchronously, the stored procedure PROCESS_DATA. The program PROCESS_DATA will run independent of the caller. Its execution does not block the caller. The following is correct but the developer should consider named notation as a means to clearify the data passed.

```
DECLARE
    id BINARY_INTEGER;
BEGIN
    dbms_job.submit(id, 'process_data;', SYSDATE);
END;
```

Even a developer familiar with the DBMS_JOB package would like to see a few formal parameter names in this case. The last argument is a built-in function, SYSDATE, and someone less familiar with DBMS_JOB might want to look at this code to see what this data means. The use of formal names makes the procedure call slightly more readable.

```
DECLARE
    id integer;
BEGIN
    dbms_job.submit(job => id, what => 'process_data;',
        next_date=> SYSDATE);
END;
```

Use named notation when you want to pick and choose specific parameters to pass—skipping some formal parameters for where there is a default value. (Default parameters are covered in Section 10.8.6, "Default Parameters."

Consider the following procedure that computes an aggregate salary. This subprogram includes default values for each parameter: MONTHLY_BASE and NO_OF_MONTHS. Calling the AGGREGATE_SALARY function with default values returns the computed salary: 10,000 times 12.

```
CREATE OR REPLACE FUNCTION aggregate_salary
    (monthly_base   NUMBER  := 10000,
     no_of_months   INTEGER := 12) RETURN NUMBER
IS
BEGIN
    return (monthly_base * no_of_months);
END;
```

To code with the default for NO_OF_MONTHS, write the following:

```
DECLARE
    monthly_base NUMBER := 9000;
    aggregate    NUMBER;
BEGIN
    -- salary for one year.
    aggregate := aggregate_salary(monthly_base);
END;
```

You cannot use positional notation and call this procedure using the default for MONTHLY_BASE and still pass the number of months. The number of months passed would be positional, interpreted as a monthly base and the result would be, for the salary, twice the number of months. Using defaults in this case requires named notation:

```
DECLARE
    no_of_months INTEGER := 10;
    aggregate    NUMBER;
BEGIN
    -- salary for 10 months.
    aggregate := aggregate_salary(no_of_months=>no_of_months);
```

10.8.6 Default Parameters

A procedure or function specification can define a default value for parameters that have IN or IN OUT mode. There are two syntax forms illustrated:

```
PROCEDURE name
    (argument mode datatype := a_default_value);

PROCEDURE name
    (argument mode datatype DEFAULT a_default_value);
```

The following function definitions each return the area of a circle with a default radius of one—the only difference between these functions is the style of the default.

```
FUNCTION circle
    (radius IN NUMBER := 1) RETURN NUMBER IS
BEGIN
    RETURN 3.14 * radius**2;
END;

FUNCTION circle
    (radius IN NUMBER DEFAULT 1) RETURN NUMBER IS
BEGIN
    RETURN 3.14 * radius**2;
END;
```

The following PL/SQL block contains four invocations of the function CIRCLE—all are valid.

```
DECLARE
   radius NUMBER := 1;
   area   NUMBER;
BEGIN
   area := circle();
   area := circle(radius);
   area := circle(radius+1);
   area := circle;
END;
```

When a subprogram contains several default parameters, the user can pick and choose any of the defaults but may have to use named notation. The following procedure updates a row in the PROFESSOR (DDL for this table is in Chapter 4). This table is:

```
SQL> desc professors
Name                           Null?    Type
------------------------------ -------- -----------
PROF_NAME (primary key)        NOT NULL VARCHAR2(10)
SPECIALTY                      NOT NULL VARCHAR2(20)
HIRE_DATE                      NOT NULL DATE
SALARY                         NOT NULL NUMBER(5)
TENURE                         NOT NULL VARCHAR2(3)
DEPARTMENT                     NOT NULL VARCHAR2(10)
```

The following update procedure requires a professor's name—this is the primary key. The caller can pass a SALARY to update and/or a TENURE (YES/NO CHECK constraint).

The purpose of this procedure is for a user to update tenure or salary. The caller passes a tenure value, passes a salary value, or passes two values—this updates both columns. The procedure returns immediately if it is called with a NULL for each paremeter.

When the default is used, the UPDATE statement uses the NVL function to set the column to the current row column value.

```
CREATE OR REPLACE PROCEDURE update_professor
   (v_prof_name professors.prof_name%TYPE,
    v_salary    professors.salary%TYPE := NULL,
    v_tenure    professors.tenure%TYPE := NULL)
IS
BEGIN
    IF
        v_salary IS NULL AND v_tenure IS NULL
    THEN
        RETURN; -- nothing to change
    END IF;

    UPDATE professors
    set    salary = NVL(v_salary, salary),
           tenure = NVL(v_tenure, tenure)
    WHERE  prof_name = v_prof_name;
END;
```

A caller can use positional notation except when they are using the default for SALARY but wish to supply a new TENURE. The following PL/SQL block contains several valid examples of calling UPDATE_PROFESSOR. The one statement that must use positional notation is on line 11.

```
 1  DECLARE
 2      new_salary  professors.salary%TYPE := 13000;
 3      new_tenure  professors.tenure%TYPE := 'YES';
 4  BEGIN
 5      -- Named notation examples
 6      update_professor('Blake', 12000, 'YES');
 7      update_professor('Blake', 12000);
 8      update_professor('Blake', new_salary, new_tenure);
 9
10      -- Positional notation examples.
11      update_professor('Blake', v_tenure => new_tenure);
12
13      update_professor('Blake', v_salary => new_salary,
14                                v_tenure => new_tenure);
15
16      update_professor(v_prof_name=> 'Blake',
17                       v_salary => new_salary,
18                    v_tenure => new_tenure);
19  END;
```

When coding procedures and functions, place defaults last in the calling sequence. The preceding example illustrates one case that requires positional notation—line 11. Defaults intermixed with nondefault parameters, or placed left of nondefault arguments, increase the number of scenarios where the user must code to named notation.

10.8.7 Scaling Code with Defaults

You can enhance the functionality of existing code by adding defaults to the end of a procedure or function specification. Because the defaults are to the right, all existing code is not affected. The existing code, as written, will use the default value assigned. New code may or may not use the parameter. New code can be written to pass a value of the default.

Consider the following procedure. It updates a professor record. Within an application there could be wide use of this procedure, yet we can extend it functionally without impacting existing code.

```
CREATE OR REPLACE PROCEDURE update_professor
   (v_prof_name professors.prof_name%TYPE,
    v_salary    professors.salary%TYPE := NULL,
    v_tenure    professors.tenure%TYPE := NULL);
```

For this scenario, assume the procedure must be modified to support email notification. To implement this we add a parameter, V_NOTIFY. This flag, when set to true, will send an email to the professor that the salary has changed. The following shows the changed code. A parameter has been added that has a default.

```
CREATE OR REPLACE PROCEDURE update_professor
   (v_prof_name professors.prof_name%TYPE,
    v_salary    professors.salary%TYPE := NULL,
    v_tenure    professors.tenure%TYPE := NULL,
    v_notify    BOOLEAN := FALSE);
```

With this change, old code is not affected. New code can pass a TRUE/FALSE value or not pass anything, using the default. For this scenario, there are two options to consider.

1. The old code plays no part in the new functionality. Perform the following steps if only new coded will use this feature.
    - Set the default for V_NOTIFY to FALSE.
    - Modify UPDATE_PROFESSORS to send an email when the V_NOTIFY is TRUE.
    - Ensure that all new code specifies a TRUE/FALSE (or default) value depending on whether an email should be sent.
    - There is no impact to existing code.
2. The old code, or some of it, will use the new functionality. To make this adjustment, do the following.
    - Set the default for V_NOTIFY, the most common situation. For example, assume most old code will send an email. In this case, set default to TRUE.
    - Modify UPDATE_PROFESSORS to send an email when the V_NOTIFY is TRUE.
    - Ensure that all new code specifies a TRUE (or default)/FALSE value in the call to UPDATE_PROFESSORS.
    - Modify the old code, which will not send an email and add a new FALSE argument in the call to UPDATE_PROFESSOR. Most old code will not change because it will use the default of TRUE.

10.8.8 %TYPE

This %TYPE syntax is a means to declare a variable whose type is derived from a particular column type in a database table. The syntax for this type definition is:

```
variable_name  table_name.column_name%TYPE;
```

Procedures that return data to a calling program must be concerned with the size or dimension of the variable to which they are writing. When the called program writes data that is larger than what the caller is expecting, the result is an overflow condition. Declaring variables using a %TYPE syntax can minimize this problem.

Consider the procedure that returns an OUT mode variable, a professors salary.

```
CREATE OR REPLACE PROCEDURE get_professor_salary
   (v_prof_name IN professors.prof_name%TYPE,
    v_salary    OUT professors.salary%TYPE);
```

Now consider the PL/SQL that uses this procedure. It must declare a salary variable. What should that type be? The following is not a good choice:

```
DECLARE
    SAL NUMBER(5,2);
BEGIN
    get_professor_salary('Milton', SAL);
END;
```

This code block will fail if the dimension of the salary column in the professors table increases. Suppose the salary recorded in the table changes from a monthly to an annual salary. The column dimension in the database would increase. Execution of the block would fail with a VALUE_ERROR exception. This would occur as the GET_PROFESSOR_SALARY attempts to write a number larger than NUMBER(5,2).

The preceding block should be written as:

```
DECLARE
    SAL professors.salary%type
BEGIN
    get_professor_salary('Milton', SAL);
END;
```

Using %TYPE with procedure and function specifications provides additional information to the user. It conveys specific information about the data expected. A procedure that updates a student record could be defined with the following:

```
PROCEDURE update_student(student IN VARCHAR2, etc);
```

The user of this procedure has to dig into the code to see whether the parameter is a student name of a student ID. Someone familiar with the database schema might understand that the parameter must be a primary key and, therefore, this procedure requires a STUDENT_ID. The mystery of the interface is easily eliminated with %TYPE.

```
PROCEDURE update_student
    (student IN students.student_ID%TYPE, etc);
```

You should not use a %TYPE when performing an aggregation. If the result of a subprogram call is the sum of all salaries, the computed number could exceed the precision of the %TYPE. The following returns a NUMBER with no precision or scale.

```
CREATE OR REPLACE PROCEDURE get_professor_salaries
   (salaries OUT NUMBER) IS
BEGIN
    SELECT SUM(salary) INTO salaries FROM professors;
END get_professor_salaries;
```

Two Oracle exceptions can occur when a datatype does not match the assigned data.

VALUE_ERROR	This exception is raised in PL/SQL when the size of the target is smaller than needed. You get this when you assign 100 to a variable declared with NUMBER(2). This is raised when you assign a 3-character string, such as "100," to a field declared as NUMBER(2). If you SELECT INTO a variable with too small a precision, you get this error.
INVALID_NUMBER	This exception can occur when a SQL statement selects character data into a NUMBER type. This implicit conversion will fail if the character string does not represent a valid number.

10.9 Overloading

You can overload the procedure or function in a package. The best example of overloading is the Oracle built-in DBMS_OUTPUT package. It provides a print procedure to display a string:

```
PUT_LINE(parameter VARCHAR2)
```

Other types can be printed including a DATE and NUMBER type. This leads to overloaded procedures:

```
PUT_LINE(parameter VARCHAR2)
PUT_LINE(parameter DATE)
PUT_LINE(parameter NUMBER)
```

You overload procedures when identical functionality is performed on different types. You can technically overload procedures that perform unrelated functions. This practice makes code that uses your package vague, hard to read, and hard to understand.

Subprograms within a package can be overloaded only when the parameter profile is different. Oracle must be able to determine which procedure you call. The following package contains overloaded procedures. This example is not valid because the base type of each parameter is the same.

```
CREATE OR REPLACE PACKAGE sample_pkg IS
   PROCEDURE process(v_value IN NUMBER);
   PROCEDURE process(v_value IN INTEGER);
END sample_pkg;
```

You can successfully compile this package but Oracle will not know which procedure to invoke if you make the following call.

```
sample_pkg.process(2);
```

This statement will fail with the Oracle message:

```
PLS-00307: too many declarations of 'PROCESS' match this call.
```

A modification to the SAMPLE_PKG package corrects this problem. The following code defines two procedures, each with a unique profile. This is a valid example of overloading.

```
CREATE OR REPLACE PACKAGE sample_pkg IS
   PROCEDURE process(v_value NUMBER);
   PROCEDURE process(v_value INTEGER, v_date DATE);
END sample_pkg;
```

Differences in MODE do not constitute differences in profile. The following is invalid.

```
CREATE OR REPLACE PACKAGE sample_pkg IS
   PROCEDURE process(v_value IN  NUMBER);
   PROCEDURE process(v_value OUT NUMBER);
END sample_pkg;
```

Differences in formal parameter names do not make the procedures unique. The following will compile but the caller must use named notation. Logically, this conflicts with the concept of overloading, which is to be able to perform similar operations of different datatypes.

```
CREATE OR REPLACE PACKAGE sample_pkg IS
   PROCEDURE process(v_new_salary IN NUMBER);
   PROCEDURE process(v_old_salary IN NUMBER);
END sample_pkg;
```

For this package, the first call works, but the second does not. Oracle will raise a run-time error because it cannot determine which procedure you want to call.

```
sample_pkg.process(v_new_salary=>9000);
sample_pkg.process(9000);
```

Overloading works when the parameter profiles are different, which includes the evaluation of the base types in conjunction with default parameters declared in the specification.

The following is a case where adding a default to an overloaded procedure breaks the code. Modify SAMPLE_PKG and add a DATE parameter with a default, showing this change as:

```
CREATE OR REPLACE PACKAGE sample_pkg IS
    PROCEDURE process(v_value NUMBER);
    PROCEDURE process(v_value INTEGER,
                      v_date DATE := SYSDATE);
END sample_pkg;
```

The package is now a problem. A call to the procedure process using the default date will fail. The following procedure call cannot be resolved. Oracle cannot determine if the user intends to call the first procedure or the second procedure using the default.

```
sample_pkg.process(2);
```

This call produces the same error—too many declarations.

```
PLS-00307: too many declarations of 'PROCESS' match this call.
```

Code reviews, even informal reviews, are opportunities for feedback on the reasonableness of overloading within a package. The DBMS_OUTPUT package is always an excellent example to ponder when designing overloaded procedures.

Chapter Eleven. PL/SQL Language Features

11.1 Comments

The double hyphen (--) is used to comment a single line. A slash followed by an asterisk (/*) begins a block comment. An asterisk followed by a slash (*/) ends a block comment.

The C-style comments (/*, */) can be used to block out a section of code. The following procedure illustrates both forms.

```
-- Filename hello.sql
CREATE OR REPLACE PROCEDURE hello IS
    str CONSTANT VARCHAR2(15) := 'Hello World!';
    len INTEGER; /* length of the constant */
BEGIN
    --
    -- Set len to length of character string.
    --
    len := LENGTH(str);

    /*
    * write the string length
    */
    dbms_output.put_line(str||'-string length:'||len);
END hello;
```

Place comments after the CREATE statement, not before. This only applies if you are using a text editor with a tool like SQL*Plus—a tool such as JDeveloper will not permit you to place comments prior to a CREATE statement. The situation to avoid is the following:

```
-- This procedure prints hello.
CREATE OR REPLACE PROCEDURE hello IS
    str CONSTANT VARCHAR2(10) := 'Hello World!';
    . . . etc
```

Comments that precede the CREATE statement are not included with the source code saved in the data dictionary. Comments after the CREATE clause are stored in the data dictionary.

Packages require extensive comments, mostly because they typically have many procedures and functions that provide a wealth of functionality. A package should have overview comments that describe the package. This overview should document any Oracle privileges required to use the package. It should also document the need for any database parameter settings.

Sometimes there is an ordered dependency among the subprograms in the package. For example, to use a package properly, you may have to call one subprogram first, prior to calling other procedures. You want to document this dependency.

An overview that includes examples that demonstrate how to use the package is very useful. These overview comments immediately follow the CREATE PACKAGE statement. Following the general comments are the procedure and function definitions, which have individual comments.

```
PACKAGE students_pkg is
--
-- general comments about how to use the package
--
    --
    -- comments on print_name
    PROCEDURE print_name (v_student_id IN VARCHAR2);
    --
    -- comments on print_major
    PROCEDURE print_major (v_student_id IN VARCHAR2);
END students_pkg;
```

The Oracle built-in packages are well documented with an overview section, frequently including examples, plus detailed comments on each procedure and function.

Packages you can use are available through the ALL_SOURCE view. This view contains the source you have written, as well as code in other schemas to which you have access. This includes the Oracle built-in packages. The interface and comments that describe the DBMS_OUTPUT package can be extracted from the ALL_SOURCE view with a SQL statement similar to that used to extract the HELLO source and comments.

```
SQL> SELECT text FROM all_source
  2  WHERE name='DBMS_OUTPUT'
  3  AND type='PACKAGE';
```

11.2 Assignments and Statements

The assignment operator is:

```
":="
```

The assignment operator is not "=", but the if-test expression is.

```
IF (a = b)
```

We do not code:

```
IF (a == b)
```

You can assign literal values, or expressions that include operations and other variables. You can wrap expressions provided the statement does not break in the middle of a variable name.

```
base_monthly_salary  := 10000;
base_salary_increase := 1000;
new_professor_salary := base_monthly_salary * 1.1;
new_profrssor_salary := base_monthly_salary +
                        base_salary_increase;
```

You can declare a variable with an initialized value. You can initialize it with NULL. Do not assume any initial value of a variable that is declared with no initial setting. You can declare an initialized variable and constrain it to NOT NULL. The following PL/SQL declares several variables.

```
DECLARE
    total_credits      NUMBER(3,1);
    total_credits_2    NUMBER(3,1) := NULL;
    credit_hour        NUMBER(2,1) := 3.0;
    no_credit CONSTANT NUMBER(2,1) := 0;
    pass_fail_credit   NUMBER(2,1) NOT NULL := 0;
BEGIN
    PL/SQL code here.
END;
```

TOTAL_CREDITS	This is a basic declaration of a NUMBER for three digits: two to the left of the decimal point, one place to the right.
TOTAL_CREDITS_2	Same as TOTAL_CREDITS except the variable is initialized to NULL.
CREDIT_HOUR	This variable has a default assignment. The value can change anytime within the code block.
NO_CREDIT	This is a constant. Constants are a form of self-documenting code. When you use a constant, you are documenting the fact that this variable is to have no other value.
PASS_FAIL_CREDIT	If you declare a variable as NOT NULL, then you must assign a default value in the same declaration.

You can code a NULL statement. For example, the following is a valid PL/SQL block and when executed, declares variables and exits.

```
DECLARE
    total_credits      NUMBER(3,1);
    total_credits_2    NUMBER(3,1) := NULL;
    credit_hour        NUMBER(2,1) := 3.0;
    no_credit CONSTANT NUMBER(2,1) := 0;
    pass_fail_credit   NUMBER(2,1) NOT NULL := 0;
BEGIN
    NULL;
END;
```

A common use of the NULL statement is within an exception handler where you want to catch the exception but intend no action. For example:

```
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN NULL;
END;
```

11.3 Boolean Expression

Boolean expressions use the operators:

- AND
- OR
- NOT

A Boolean expression can evaluate to TRUE, FALSE, or possibly not evaluate to anything. An expression does not evaluate when it contains a NULL—this topic is covered in the next section, "Expressions with NULL."

Lengthy Boolean expressions can be difficult and time consuming to interpret. You can lessen the impact of hard-to-read logic if you wrap long Boolean expressions into a PL/SQL function that returns a BOOLEAN.

For example, within the body of a PL/SQL procedure, it could take some time to digest the following logic.

```
If NOT ((status = 'PART TIME' OR status = 'FULL TIME')
    AND (balance = 0)
    AND NOT (college_major = 'Undeclared')) THEN
```

The same logic, when embedded in a function, simplifies to the code. The following procedure encapsulates the Boolean logic into a function by the name:

```
registered_student_with_no_credit().
```

This is a long function name, but when used in the body of the PL/SQL program, conveys the Boolean test being performed.

```
PROCEDURE process_student_registration (. . .)
IS
    -- Local function that encapsulates this logic.
    FUNCTION registered_student_with_no_credit RETURN BOOLEAN
IS
    BEGIN
       RETURN NOT
        ((status = 'PART TIME' OR status = 'FULL TIME')
          AND (balance = 0)
          AND NOT (college_major = 'Undeclared'));
    END registered_student_with_no_credit;
BEGIN
    . . .
    body of procedure
    . . .
    IF registered_student_with_no_credit THEN
        Do something . . .
    ELSE
        Do other thing
    END IF;
END process_student_registration;
```

A programmer can look at the body of this procedure and get an idea of what the code does. A general understanding is that the code looks at students without credit. Logic is executed based on that test. The complex Boolean logic is still in the procedure, localized to a function. The programmer can dig into that logic if necessary. The encapsulation of the Boolean code allowed the procedure to be reviewed in steps.

11.4 Expressions with NULL

The combination of a NULL with an AND in a predicate will not give you the results you expect. A variable that is NULL has no value. Therefore, you cannot compare it to anything else with meaningful results. An expression of comparison with a NULL does not evaluate. When such an expression exists within an IF statement and there is an ELSE part, the code does not evaluate to TRUE, so it follows the ELSE path. From this, it can appear that the condition evaluates to FALSE. If you then reverse the logic in the IF test with a NOT operator, the same ELSE path is taken.

Consider two variables: OTHER_NAME and MY_NAME. A NULL initializes the variable OTHER_NAME.

```
my_name     VARCHAR2(100) := 'SCOTT';
other_name  VARCHAR2(10) := NULL;
```

The following expression does not evaluate:

```
IF (my_name = 'SCOTT' AND other_name = 'JOHN') THEN
```

Because this expression does not evaluate, it cannot possibly evaluate to TRUE. Hence, the following PL/SQL will follow the ELSE path.

```
set serveroutput on
DECLARE
    my_name     VARCHAR2(10) := 'SCOTT';
    other_name  VARCHAR2(10) :=  NULL;
BEGIN
    IF (my_name = 'SCOTT' AND other_name = 'JOHN') THEN
        dbms_output.put_line('CONDITION_TRUE');
    ELSE
        dbms_output.put_line('CONDITION_FALSE');
    END IF;
END;
```

The result of this script is the following (notice the execution path follows the ELSE part).

```
CONDITION_FALSE
PL/SQL procedure successfully completed.
SQL>
```

Based on this outcome, the opposite of the IF condition should take the opposite execution path. That is, if we put a NOT before our test, then the code should NOT take the ELSE path and display CONDITION_TRUE. The following PL/SQL block is changed. The only change is to place a NOT in the IF statement.

```
set serveroutput on
DECLARE
    my_name     VARCHAR2(10) := 'SCOTT';
    other_name  VARCHAR2(10) :=  NULL;
BEGIN
    IF NOT (my_name = 'SCOTT' AND other_name = 'JOHN') THEN
        dbms_output.put_line('CONDITION_TRUE');
    ELSE
        dbms_output.put_line('CONDITION_FALSE');
    END IF;
END;
```

The result of this script is the same.

```
CONDITION_FALSE
PL/SQL procedure successfully completed.
SQL>
```

Based on the preceding two PL/SQL blocks, the following is never true

```
(my_name = 'SCOTT' AND other_name = 'JOHN')
```

The following is also never true:

```
NOT (my_name = 'SCOTT' AND other_name = 'JOHN')
```

The conclusion from this is that PL/SQL conditions with an AND part and a NULL expression do not evaluate. They do not evaluate to TRUE; they do not evaluate to FALSE. If there is an ELSE part in the IF statement, the code will take that path.

A condition with the OR operator that includes at least one test that evaluates to TRUE will behave as expected. The following expression displays CONDITION_TRUE because, at least, the first part, MY_NAME=SCOTT is TRUE.

```
set serveroutput on
DECLARE
    my_name     VARCHAR2(10) := 'SCOTT';
    other_name  VARCHAR2(10) :=  NULL;
BEGIN
    IF (my_name = 'SCOTT' OR other_name = 'JOHN') THEN
        dbms_output.put_line('CONDITION_TRUE');
    ELSE
        dbms_output.put_line('CONDITION_FALSE');
    END IF;
END;
```

This output is TRUE because at least MY_NAME=SCOTT is true.

```
CONDITION_TRUE
PL/SQL procedure successfully completed.
SQL>
```

When all parts of an OR condition do not evaluate, the entire expression does not evaluate. In this case, the statement is neither TRUE nor FALSE. The IF test fails to be TRUE because it does not evaluate. This takes the code to the ELSE part and leads one to believe that the condition does not evaluate to TRUE and must be FALSE. This is the case for the following:

```
set serveroutput on
DECLARE
    my_name     VARCHAR2(10) := 'SCOTT';
    other_name  VARCHAR2(10) :=  NULL;
BEGIN
    IF (other_name = 'JOHN' OR other_name = 'SCOTT') THEN
        dbms_output.put_line('CONDITION_TRUE');
    ELSE
        dbms_output.put_line('CONDITION_FALSE');
    END IF;
END;
```

Because no part of the OR test evaluates, the entire test does not evaluate. The result of the preceding block is the following output.

```
CONDITION_FALSE
PL/SQL procedure successfully completed.
SQL>
```

Reversing the logic of the IF test does not reverse the execution path, as shown next.

```
set serveroutput on
DECLARE
    my_name     VARCHAR2(10) := 'SCOTT';
    other_name  VARCHAR2(10) :=  NULL;
BEGIN
    IF NOT (other_name = 'JOHN' OR other_name = 'SCOTT') THEN
        dbms_output.put_line('CONDITION_TRUE');
    ELSE
        dbms_output.put_line('CONDITION_FALSE');
    END IF;
END;
```

The result is the same. The IF condition does not evaluate. It follows the execution of the ELSE path, shown here.

```
CONDITION_FALSE
PL/SQL procedure successfully completed.
SQL>
```

We can use the NVL operator in these situations.

NVL is a function that takes two arguments. If the first argument is NULL, the NVL function returns the second argument. For the following, NVL returns 1 when ARG is NULL.

```
NVL(arg, 1)
```

You can nest NVL functions. This allows you to select a NOT NULL value from a list of candidates. Consider variables: A, B, and C. You want A. If it is NULL, then use B. If that is NULL, then use C. If all are NULL, then use zero. The expression for this is:

```
NVL(A, NVL(B, NVL(C,0)))
```

For the following, we can replace OTHER_NAME with a blank only when it is NULL. This removes NULLs from the Boolean expression.

```
set serveroutput on
DECLARE
   my_name     VARCHAR2(10) := 'SCOTT';
   other_name  VARCHAR2(10) :=  NULL;
BEGIN
   IF
      NOT (NVL(other_name,' ') = 'JOHN' OR
          NVL(other_name,' ') = 'SCOTT')
   THEN
      dbms_output.put_line('CONDITION_TRUE');
   ELSE
      dbms_output.put_line('CONDITION_FALSE');
   END IF;
END;
```

The output from this block is:

```
CONDITION_TRUE
PL/SQL procedure successfully completed.
SQL>
```

In summary, use caution when writing PL/SQL procedures that have IN or IN OUT mode parameters and those parameters are to be embedded in IF statements with ELSE logic. Someone may call your procedure passing a NULL and the code will execute a path that makes no sense.

A strategy is to emphasize NOT NULL constraints in the schema tables. By applying NOT NULL constraints, the values read from select statements will not be NULL. A goal should be to reduce the number of NULL variables within an application.

You also have the NVL function as shown earlier. You can use the NVL operator in your SQL select statements. This can be an initial opportunity to translate a NULL to a blank or zero as you pull the data from the database.

You can replace database NULL defaults with a NOT NULL value (see Chapter 3). A SQL INSERT and UPDATE places a NULL into a column that is not referenced in the SQL statement. This is the default behavior. Consider a table column that stores a checkbook balance. Make the default a zero rather than NULL. PL/SQL code that queries checkbook data will get a zero rather than a NULL. The decision to use a zero default rather than a NULL is because the absence of money, in this case, is zero. This eliminates the concern with embedding a checkbook balance within Boolean expressions.

11.5 Logical Operators

PL/SQL supports a full set of logical operators that may be used for logical and arithmetic expressions. The operations defined on DATE type allow manipulation and comparison of DATE type variables. Use "IS NULL" and "NOT NULL" in comparison statements if any of the values may be NULL. The following is a summary of comparison operators.

```
 =    is equal to                 IF (a = b)  THEN
!=    is not equal to              IF (a != b) THEN
<>     is not equal to              IF (a <> b) THEN
>     is greater than              IF (a = b)  THEN
>=     is greater than or equal to   IF (a != b) THEN
<     is less than                IF (a <> b) THEN
<=     is less than or equal to     IF (a != b) THEN
```

The following function returns TRUE if the DATE parameter passed is yesterday.

```
FUNCTION is_yesterday (v_date in DATE) RETURN BOOLEAN IS
BEGIN
    RETURN (TRUNC(v_date) = TRUNC(SYSDATE-1));
END is_yesterday;
```

This function can be used in conjunction with operators.

```
IF (a < b) AND is_yesterday(date_variable) THEN
```

You can also use the following for comparisons.

- BETWEEN
- LIKE

  % For string substitution

  _ For a single character substitution

- IN

To test if a student course name contains the string "MATH," use string substitution on either side. The following returns TRUE if MATH is anywhere in the variable.

```
IF (variable_name LIKE '%MATH%') THEN
```

What if the variable name is set to "Math and Science." The test fails because the string "MATH" is not the same as "Math." When not sure about case, convert the variable to upper or lower case and then do the comparison.

```
IF (UPPER(variable_name) LIKE '%MATH%') THEN
```

The following returns TRUE when the uppercase variable has an "E." Any single character follows that "E." And a "B" follows that single character.

```
IF (UPPER(variable_name) LIKE '%E_B%') THEN
```

When variable_name = EEB this test is true. When variable_name = abcexbx this test is true.

Comparisons do include endpoints. The following is TRUE if VARIABLE equals MIN_VALUE of MAX_VALUE.

```
IF (variable BETWEEN min_value AND max_value) THEN
```

The IN operator returns TRUE if a variable is found in a set.

```
IF (variable IN ('BOSTON', 'CHICAGO','LONDON')) THEN
```

The IN parameters can be variables. For example:

```
CREATE OR REPLACE procedure test(arg VARCHAR2)
IS
    chicago VARCHAR2(10) := 'CHICAGO';
    new_york VARCHAR2(10) := 'NEW_YORK';
BEGIN
    IF (arg IN (chicago, new_york)) THEN
        Other code
END;
```

Use SQL functions to reduce unnecessary comparison logic in PL/SQL. This applies to comparing different column values from a single row. Suppose you select two columns and ultimately want the larger of the two. One approach is to code the following:

```
SELECT COL_1, COL_2 INTO VAR_1, VAR_2 FROM etc.
IF  VAR_1 > VAR_2 THEN etc.
```

The SQL functions GREATEST, LEAST can assist.

```
SELECT COL_1, COL_2,
       GREATEST(COL_1, COL_2) INTO VAR_1, VAR_2, VAR_3 etc.
```

Refer to Section 11.14, "Miscellaneous String Functions," for a description of GREATEST and LEAST—they also operate on strings.

[Team LiB]

11.6 String Concatenation

The concatenation operator is:

```
||
```

This operator concatenates strings, but you can concatenate other types. PL/SQL does implicit conversion, such as converting a number and date type variable to a string. This enables statements to concatenate various types, but only if the concatenated item converts to a string.

```
DECLARE
    professor_name VARCHAR2(100) := 'Professor Smith';
    hire_date      DATE          := SYSDATE;
    dalary         NUMBER(7,2)   := 10100.50;
    str            VARCHAR2(2000);
BEGIN
    str := professor_name|| ' was hired on '||
           hire_date ||' with a salary of '||
           salary||' per month. ';
    dbms_output.put_line(str);
END;
```

You precede each single quote that is part of the string with a quote. For example, your string needs to include a single quote because the final text must be this next line:

```
A quote like ' is needed.
```

You produce this output by preceding the quote with a quote. This identifies the character as a quote in the string rather than the end of the string.

```
DECLARE
    str varchar2(100);
BEGIN
    str := 'A quote like '' is needed.';
    dbms_output.put_line(str);
END;
```

When quotes must begin or end a string, you still have two quotes. You also have the string-terminating quote. You want the output to be:

```
'A quote like ' is needed.'
```

You form this string with the following:

```
DECLARE
    str varchar2(100);
BEGIN
    str := '''A quote like '' is needed.''';
    dbms_output.put_line(str);
END;
```

The SQL*Plus environment uses "&" for parameter notation. When you write PL/SQL in a SQL*Plus environment, the use of this symbol and the text that follows it will be interpreted as a SQL*Plus parameter. This is only an issue in SQL*Plus. To use & in a string, replace it with the CHR function. The CHR function is a built-in function that returns the ASCII character from an equivalent ASCII number. The ASCII integer for & is 38. Refer to Section 11.14, "Miscellaneous String Functions," for a description of CHR and ASCII. If our output is:

```
'A quote like ' and a & is needed.'
```

You can display this with the following.

```
DECLARE
    str varchar2(100);
    ch  varchar2(1) := CHR(38);
BEGIN
    str := '''A quote like '' and a '||ch||' is needed.''';
    dbms_output.put_line(str);
END;
```

11.7 Arithmetic Expressions

All third-generation language mathematical operations are provided.

```
x := -1;           -- x equals minus 3
y := -x;           -- y equals 3
y := -((-3))*(-1)) -- y equals –3
x := 2**10;        -- x equals 1024
x := 1/((60/24)    -- x equals number of seconds in a day
```

11.8 Variable Declarations

You declare local subprogram variables in the declarative region between the keywords: IS and BEGIN.

```
PROCEDURE procedure_name(. . .)
IS
    constants, types, variables declare here.
BEGIN
    body of code.
END procedure_name;

FUNCTION function_name(. . .) RETURN some_datatype
IS
    constants, types, variables declare here.
BEGIN
    body of code.
    RETURN value_to_return;
END function_name;
```

The scope of a procedure or function variable is only the subprogram in which the variable is declared. The following is a package specification and body with two procedures. Each procedure has a single variable. Each procedure is autonomous. The variables NAME and MAJOR are local to their respective subprograms.

```
PACKAGE students_pkg is
    PROCEDURE print_name (v_student_id IN VARCHAR2);
    PROCEDURE print_major (v_student_id IN VARCHAR2);
END students_pkg;

PACKAGE BODY students_pkg IS
    PROCEDURE print_status (v_student_id IN VARCHAR2)
    IS
        name VARCHAR2(100);
    BEGIN
        SELECT student_name INTO name
          FROM STUDENTS
         WHERE student_id = v_student_id;
        dbms_output.put_line(name);
    END print_status;
    PROCEDURE print_major (v_student_id IN VARCHAR2)
    IS
        major VARCHAR2(100);
    BEGIN
        SELECT college_major INTO major
          FROM STUDENTS
         WHERE student_id = v_student_id;
        dbms_output.put_line(major);
    END print_major;
END students_pkg;
```

You can declare variables in the declarative part of the package body. This makes the variables global to all procedures and functions within the package body. The preceding package is different only with respect to the placement of the variables NAME and MAJOR.

```
PACKAGE BODY students_pkg IS
    name VARCHAR2(100);
    major VARCHAR2(100);

    PROCEDURE print_status (v_student_id IN VARCHAR2)
    IS
    BEGIN
        Same code as above.
    END print_status;
    PROCEDURE print_major (v_student_id IN VARCHAR2)
    IS
    BEGIN
        Same code as above.
    END print_major;
END students_pkg;
```

As with any language, excessive use of global variables makes tracing and troubleshooting difficult.

You can declare variables in the package specification. The following is a simple package that will compile and stand on its own within an application environment. If the user SCOTT compiles this package then all of SCOTT's PL/SQL programs will be able to reference the state of this variable or change it at any time.

```
PACKAGE global_pkg IS
    my_global_variable INTEGER := 0;
END global_pkg;
```

You should encapsulate the global variable with procedures and functions that set and return the value of the variable, shown next (refer to Chapter 10 for discussion of information hiding).

```
PACKAGE global_pkg IS
    PROCEDURE set_value(new_value INTEGER);
    FUNCTION current_value RETURN INTEGER;
END global_pkg;

PACKAGE BODY global_pkg IS
    my_global_variable INTEGER := 0;
    PROCEDURE set_value(new_value INTEGER) IS
    BEGIN
        my_global_variable := new_value;
    END set_value;
    FUNCTION current_value RETURN INTEGER IS
        RETURN my_global_variable;
    BEGIN
    END get_value;
END global_pkg;
```

11.9 Types

11.9.1 Boolean

You can declare a BOOLEAN PL/SQL variable type. There is no BOOLEAN database column type.

Booleans are easy to work with. Often the existence of a value logically relates to a Boolean variable. For the student's demo, a student can have several majors (e.g., Biology). One valid lookup value is: Undeclared. Suppose you consider Undeclared to be FALSE and any other major to be TRUE. Logically this equates to: having a major is TRUE. Having an undeclared major is false. Such a scenario can be common in application code.

It helps to have a package that maps Boolean values to a 1 and 0. For example, the following package is a general purpose package that transforms a Boolean to and from a zero or one.

```
CREATE OR REPLACE PACKAGE bool IS
    FUNCTION to_int (B BOOLEAN) RETURN NATURAL;
    FUNCTION to_bool(N NATURAL) RETURN BOOLEAN;
END bool;

CREATE OR REPLACE PACKAGE BODY  bool IS
    FUNCTION to_int (B BOOLEAN) RETURN NATURAL IS
    BEGIN
        IF B THEN RETURN 1; ELSE RETURN 0; END IF;
    END to_int;
    FUNCTION to_bool(N NATURAL) RETURN BOOLEAN IS
    BEGIN
        IF N=1 THEN RETURN TRUE; ELSE RETURN FALSE; END IF;
    END to_bool;
END bool;
```

A demonstration of the preceding package is a query to get the first major from the MAJOR_LOOKUP table and saves a 1 or 0 using a DECODE statement. The NATURAL number flag is converted to a Boolean using the package.

```
DECLARE
    major            NATURAL;
    a_declared_major BOOLEAN;
BEGIN
    SELECT DECODE (major_desc, 'Undeclared', 0, 1)
      INTO major
      FROM major_lookup WHERE ROWNUM = 1;

    a_declared_major := bool.to_bool(major);
END;
```

11.9.2 Scalar Types

You should declare PL/SQL variable types from table column types whenever possible. This makes the code flexible. It is also a form of self-documenting code. The following procedure has one parameter. It is a variable-length string.

```
PROCEDURE process(arg IN VARCHAR2);
```

The next procedure is a small improvement and uses a more meaningful formal parameter name.

```
PROCEDURE process(state IN VARCHAR2);
```

This is best because it suggests that the parameter passed is to be a value with the dimension of the STATE_DESC column in the STATE_LOOKUP table.

```
PROCEDURE process(state IN state_lookup.state_desc%TYPE);
```

There is no impact on the PL/SQL code if the dimension of the STATE_DESC should change. If the column STATE_DESC in the table STATE_LOOKUP is altered, the PL/SQL code will be recompiled, but this is minor compared to editing many PL/SQL programs and changing the dimension of a VARCHAR2.

As mentioned, the use of %TYPE provides built-in documentation. The STATE_LOOKUP table has two columns, each are VARCHAR2. Using the %TYPE makes it clear which column value the procedure is expecting.

Using %TYPE also aids the process of impact analysis. Suppose you want to change the STATE_LOOKUP table. The USER_DEPENDENCIES view will show this procedure as being dependent on the STATE_LOOKUP table. Because the procedure merely references the column with a %TYPE attribute, the procedure is formally dependent on this table. Using %TYPE will enhance impact analysis any time you need to assess what code will be affected with a table change.

Some common PL/SQL predefined types include the following:

CHAR

A CHAR variable stores a constant-length string. An optional length denotes the number of bytes with a maximum of 32767 bytes. Subtypes are STRING and CHARACTER.

```
variable  CHAR      -– stores a single character.
variable  CHAR(10); -- stores 10 characters
```

The Oracle error ORA 6502 is raised if you assign a string that exceeds the dimension of the CHAR variable. This error ORA 6502 is mapped to an exception VALUE_ERROR.

Assigning a string less than the dimension right pads the variable with blanks.

Keep in mind that you can right pad a VARCHAR2 variable with the RPAD function.

```
DECLARE
    C CHAR(3);        -- maximum of 3 characters
    X VARCHAR2(30);
BEGIN
    C := 'ABCDE';     -- raises VALUE_ERROR
    C := 'A';         -- C is: A and 2 blanks

    X := RPAD('A', 3);  -- X is: A and 2 blanks,
                        -- also X is equal to C
EXCEPTION
    WHEN VALUE_ERROR THEN
        dbms_output.put_line('Value error');
END;
```

VARCHAR2

A VARCHAR2 stores a variable length string. The maximum is 4000 bytes. Most data in a database is character data. Use %TYPE when declaring PL/SQL variables that are populated with database data. Otherwise use VARCHAR2. You can store larger character strings with the CLOB datatype.

There are many string functions that support manipulation of VARCHAR2 types. See Section 11.13, "String Manipulation Functions."

NUMBER

Use NUMBER to store fixed or floating-point numbers. You specify the precision with the notation:

```
NUMBER[(precision, scale)]
```

Precision is the total number of decimal places.

Scale is the number of those places to the right of the decimal point. A scale can be negative. This indicates significance of digits to the left of the decimal point. If you want to store 5-digit numbers and always store the number rounded to hundredths, the precision is 3 and the scale is –2.

NUMBER(3,-2)	This has a format of 99900.

Numbers are rounded to fit the precision. A value too large will raise a VALUE_ERROR exception.

To declare a type test score that includes two decimal places to the right and with numbers that range from zero to 100:

NUMBER(5,2)	This has a number format 999.99.
NUMBER(3)	This stores any three-digit whole number.
NUMBER(1,-2)	This stores 100, 200, up to 900.

You can control rounding when inserting values to a NUMBER type. There are a variety of powerful functions: ROUND, TRUNC, CEIL, FLOOR. Refer to Section 11.15, "Numeric Functions."

DATE

A DATE type stores the DAY and TIME in a single variable. Refer to Section 11.17, "Date Functions," for DATE manipulation.

Use integer-based numbers when there is no fractional part. Using a NUMBER type to store basic integers requires additional space. NUMBER types use floating precision that is unnecessary for integer variables.

PLS_INTEGER	Use this to store whole numbers. The range is: –2**31 to 2**31
BINARY_INTEGER	This is another integer but performance is better with PLS_INTEGER types.
POSITIVE	Use this to store numbers 1 and greater. A VALUE_ERROR exception is raised if you assign a number less than 1.
NATURAL	Use this to store numbers zero and greater. A VALUE_ERROR exception is raised if you assign a number less than zero.

11.9.3 Records

Records are composite structures that allow multiple pieces of data to be passed as a single parameter. Rather than pass name, birth date, and age to a procedure, pass a single record with the components: name, birth date, and age.

You must first declare a record type definition; then declare an object of that type.

You declare the record type with the syntax:

```
TYPE record_type_name IS RECORD(
component_1 component_1_type [NOT NULL := expression],
component_2 component_2_type [NOT NULL := expression],
component_3 component_3_type [NOT NULL := expression]);
```

Common scalar component types include DATE, NUMBER, and VARCHAR2. To declare a record type using these scalar types:

```
TYPE record_type IS RECORD(
   name       VARCHAR2(30),
   address    VARCHAR2(30),
   birth_date DATE,
   age        NUMBER(3));
```

A record component type can be derived from a column type. For example:

```
TYPE record_type IS RECORD(
   name    students.student_name%TYPE,
   major   major_lookup.major_desc%TYPE);
```

You declare a record structure based on the record type. The record declaration comes after the type definition. The following declares a record type, then fetches into that record. The block ends with printing the components of the record.

```
DECLARE

   TYPE record_type IS RECORD(
   name     students.student_name%TYPE,
   major    major_lookup.major_desc%TYPE);

   student_rec record_type;

BEGIN

  SELECT student_name, major_desc
    INTO student_rec
    FROM students a, major_lookup b
   WHERE student_id='A101' AND a.college_major=b.major;

   dbms_output.put_line(student_rec.name);
   dbms_output.put_line(student_rec.major);
END;
```

You address record components using a dot to reference the components. For example:

```
student_rec.name := 'Ricky';
student_rec.major := 'Biology';
```

Records can be assigned to one another provided they are derived from the same record type definition.

You can preset default component values in the type definition. The following declares a record type and presets each component value. All record declarations using this type have each component initialized to the current day and zero, respectively.

```
TYPE record_type IS RECORD(
    birth_date DATE      := TRUNC(SYSDATE),
    age        NUMBER(3) := 0);
```

The type definition initializes BIRTH_DATE using the built-in function SYSDATE. You can use any function, including your PL/SQL function from an existing package.

The ideal use of a record is to reduce parameters into and out of procedures. Consider a function that must return the name and college major of a student. The following illustrates a simple interface that evolves when records are used. The function STUDENT returns a single record based on a primary key STUDENT_ID.

```
CREATE OR REPLACE PACKAGE students_pkg IS

    TYPE student_type IS RECORD(
    name     students.student_name%TYPE,
    major    major_lookup.major_desc%TYPE);

    FUNCTION student(ID IN students.student_id%TYPE)
        RETURN student_type;

END students_pkg;

CREATE OR REPLACE PACKAGE BODY students_pkg IS
    FUNCTION student (ID IN students.student_id%TYPE)
        RETURN student_type
    IS
        r student_type;
    BEGIN
        SELECT student_name, major_desc
          INTO r
          FROM students a, major_lookup b
         WHERE student_id='A101' AND
               a.college_major=b.major;

        RETURN r;
    END student;
END students_pkg;
```

The caller of the procedure can use the package function in an assignment statement. The program that calls the function can reference each component within an expression. The following block merely calls DBMS_OUTPUT to print the name and major of a particular college student—simply by referencing the returned record component.

```
BEGIN
    dbms_output.put_line(students_pkg.student('A101').name);
    dbms_output.put_line(students_pkg.student('A101').major);
END;
```

Records can be nested. A record component can be another record. A record component can be a record whose type is derived from a database table row type. For example, the following illustrates a record with two components. The first is a student ID; the second component is a record structure. The subcomponent names of STUDENT_ADVISOR are the column names of the PROFESSORS table. This could store a student ID and all relevant information about that student's academic advisor.

```
TYPE record_type IS RECORD(
    student         students.student_id%TYPE,
    student_advisor  professors%ROWTYPE);
```

The following illustrates code that creates a type with a component from the STUDENTS_PKG. This new record has two components. The first is a student ID. The second component is derived from the type declaration in the package specification.

```
DECLARE
    TYPE my_student_type IS RECORD(
        ID          VARCHAR2(10),
        student     students_pkg.student_type);
    stud my_student_type;
BEGIN
    stud.ID := 'A101';

    stud.student := students_pkg.student(stud.ID);
    dbms_output.put_line(stud.student.name);
END;
```

A record component can be an index-by table. The following declares an index-by table to contain student test scores. The record type definition has two components: student ID and a structure for test scores. A record structure, STUDENT_TEST_SCORES, encapsulates the student and text scores into a single structure. Index-by tables are covered in the following section.

```
TYPE table_type IS TABLE OF NUMBER(3) INDEX BY
BINARY_INTEGER;
TYPE student_test_scores_type IS RECORD(
    student     students.student_id%TYPE,
    test_scores table_type);

student_test_scoure student_test_scores_type;
```

11.9.4 %ROWTYPE

You can declare a record structure based on the columns of an existing table. You do not declare a type definition. Use %ROWTYPE when coding procedures and functions that perform DML on tables. A following procedure adds a student and has one record parameter.

```
CREATE OR REPLACE PACKAGE students_pkg IS
    PROCEDURE add_student(rec IN students%ROWTYPE);
END students_pkg;
```

Addressing components is the same as a record. The record component names are the table column names. The datatype of each component is the same type as the column in the table. The following references two components in the record.

```
dbms_output.put_line(rec.student_id);
dbms_output.put_line(rec.student_name);
```

The STUDENTS table must exist before a reference to students %ROWTYPE will compile.

Oracle recognizes that the package is now dependent on the STUDENTS table. One benefit to using %ROWTYPE and %TYPE is the dependency information you retrieve from the USER_DEPENDENCIES view.

11.9.5 Index-By Tables

Index-by tables are unconstrained indexed liner structures, similar to arrays. They are originally sparse. You can populate table position 20 and position 30 using only the memory required for those two slots.

Index-by tables are constructed with the following syntax:

```
TYPE table_name IS TABLE OF component_type [NOT NULL]
INDEX BY BINARY_INTEGER;
```

For example, the following declares a table type that can be used later to declare a table of DATE type variables.

```
TYPE date_table_type IS TABLE OF DATE INDEX BY BINARY_INTEGER;
```

You can constrain the elements to NOT NULL with the syntax:

```
TYPE date_table_type IS TABLE OF DATE NOT NULL
INDEX BY BINARY_INTEGER;
```

Common scalar component types include DATE, NUMBER, and VARCHAR2. The following declares an index-by table. You reference table elements with parentheses enclosing an integer or any expression that evaluates to an integer.

```
DECLARE
    TYPE table_type IS TABLE OF INTEGER INDEX BY
BINARY_INTEGER;
    tab table_type;
BEGIN
    tab(1) := 1000;
    tab(4) := 200;
END;
```

A table element can be a record structure. The following table definition derives the table element from the STUDENTS_PKG shown in Section 11.9.3, "Records." For this code, each table element is a record structure based on the STUDENT_TYPE record of the package. A student record is fetched using the package and stored into the table at slot 101.

```
DECLARE
    TYPE table_type IS TABLE OF students_pkg.student_type
    INDEX BY BINARY_INTEGER;

    tab table_type;

    student_id students.student_id%TYPE := 'A101';

    location binary_integer;
BEGIN

    location := to_number(substr(student_id,2));

    tab(location) := students_pkg.student(student_id);

    dbms_output.put_line(tab(location).name);
    dbms_output.put_line(tab(location).major);

    -- NO_DATA_FOUND
    dbms_output.put_line('d'||tab(location+1).name);
    dbms_output.put_line(tab(location).major);
END;
```

The exception NO_DATA_FOUND is raised if you read from a table location to which no data has been assigned. The NO_DATA_FOUND exception is a motivation for either packing data in sequence and keeping track of where the data is located or using the built-in table attributes. These attributes allow your code to sequence through sparse data beginning with the FIRST and ending with the LAST element. You avoid reading empty slots with the EXISTS attribute. The built-in attributes are:

• COUNT	Returns the number of locations with data.
• FIRST	Smallest integer location containing data.
• LAST	Greatest integer location with data.
• PRIOR	Returns the integer to the prior location containing data.
• NEXT	Returns the integer of the next location containing data.
• EXISTS	Guards against reading empty cells.
• DELETE	Deletes an element and frees memory.

The following illustrates table attributes. If a table is empty, FIRST and LAST evaluate to NULL. PRIOR and NEXT can also return NULL. PRIOR evaluates to NULL if you pass an integer argument whose value is before the first table index. PRIOR evaluates to NULL if you pass an integer argument whose value comes after the last table index.

The following block loads two slots. The block includes DBMS_OUTPUT calls using NEXT and PRIOR. The value displayed is shown as a comment next to each PUT_LINE call.

```
DECLARE
    TYPE table_type IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;
    tab table_type;
BEGIN
    tab(1) := 1000;
    tab(4) := 200;
    dbms_output.put_line(tab.COUNT);

    FOR I in tab.FIRST..tab.LAST LOOP

        IF tab.EXISTS(I) THEN
            dbms_output.put_line(tab(I));
        ELSE
            dbms_output.put_line('no data in slot '||I);
        END IF;
    END LOOP;

    dbms_output.put_line(tab.next(0));  -- 1
    dbms_output.put_line(tab.prior(0)); -- null
    dbms_output.put_line(tab.next(1));  -- 4
    dbms_output.put_line(tab.next(2));  -- 4
    dbms_output.put_line(tab.next(4));  -- null
    dbms_output.put_line(tab.prior(5)); -- 4
END;
```

11.9.6 Varrays and Nested Tables

Within PL/SQL, a VARRAY is a constrained array structure. Its size is constrained, whereas index-by tables are unbounded. A VARRAY is created with a maximum dimension.

You can declare index-by table types and record types in a package specification and declare procedures and functions to operate on objects of such types. This use of composite structures enhances the PL/SQL code. It means programs can pass a single record with nested components, including nested index-by tables, rather numerous parameters. You cannot store a record structure or an index-by composite structure as an object in the database. However, composite structure objects are supported. The following illustrates the use of varrays and nested tables.

The following declares a VARRAY of numbers and uses the constructor function to initialize values. First the type is defined. This is a VARRAY object type capable of storing up to four numbers. Then an object, MY_NUMBERS, of this type is declared. The maximum size is four, but less than four numbers can by assigned to the varray. COUNT is a built-in attribute that evaluates to the number of elements in the array. The varray indexing starts at 1 and is dense—the array is populated in sequence.

```
DECLARE
    TYPE my_numbers_type IS VARRAY(4) OF NUMBER;
    my_numbers my_numbers_type;
BEGIN
    my_numbers := my_numbers_type(1,2,3);
    FOR I IN 1..my_numbers.COUNT LOOP
        dbms_output.put_line(my_numbers(I));
    END LOOP;
END;
```

Scientific measurements are seldom a single number. A scientist will find it convenient if a database table element can be an (X,Y) coordinate. Start with declaring a TYPE that represents an (X,Y) coordinate. The following is the DDL that creates such a type in the database. This is a new object you will find in USER_OBJECTS. The object_name is POINT_TYPE and the object_type is TYPE.

```
CREATE OR REPLACE TYPE point_type AS OBJECT
(
    x  NUMBER,
    y  NUMBER);
```

The POINT_TYPE provides a lot of flexibility. Prior to this we would likely create a record structure with (X,Y) components and then create an index-by table type with the record type as the element stored in the table. Creation of POINT_TYPE as an object in the database will lead to storing points as single POINT_TYPE entities in the database. We won't have to create a table with a column for X and a column for Y. Rather, we will create a table with a column of type POINT_TYPE. Furthermore, we will shortly store an array of points as a single entity.

The following PL/SQL block declares a type that is a VARRAY of elements where each element is a POINT_TYPE. This type is called POINTS_VARRAY_TYPE. The variable, POINTS, is an object of this type. This is similar to the preceding PL/SQL block except that the varray consists not of numbers but (X,Y) coordinates.

POINT_TYPE is now an object in the database. Because POINT_TYPE is now defined in the database, we can declare a VARRAY type of POINT_TYPE elements. We can also declare POINT_TYPE variables (in the following PL/SQL block) such as A_POINT and PT.

```
DECLARE
    TYPE   points_varray_type IS VARRAY(10) OF point_type;
    points points_varray_type := points_varray_type();
    a_point   point_type;
    pt        point_type;
BEGIN
    a_point := point_type(3,4);

    points := points_varray_type(point_type(1,2),
                                 point_type(2,3), a_point,
                                 point_type(5,4));

    FOR I IN 1..points.COUNT LOOP
        pt := points(I);
        dbms_output.put_line('x='||pt.x||' y='||pt.y);
    END LOOP;
END;
```

We can create additional database objects using the existing POINT_TYPE declaration. The following creates two more object types. The first is a varray type. The second is a nested table type. Each of the following two DDL statements creates an object in the database. Each is dependent on the existing database object, POINT_TYPE.

```
CREATE OR REPLACE TYPE points_varray_type
                       AS
                       VARRAY(10) OF point_type;

CREATE OR REPLACE TYPE points_nested_table_type
                       AS
                       TABLE OF point_type;
```

In review, we started with a point type, POINT_TYPE, and a PL/SQL-type definition that allows us to manipulate a VARRAY of points within a PL/SQL program. The varray and nested table types allow for database table creation to consist of columns that consist of point arrays.

The following table is used to store environmental measurement data. The varray and nested table are each used to illustrate the differences. The VARRAY column is stored as packed data within the table. Use a VARRAY for small amounts of data for which you know the maximum dimension. Use nested tables for larger data elements and unknown dimensions.

```
CREATE TABLE environment_data
(
 sample_ID            NUMBER(3),
 points_varray        points_varray_type,
 points_nested_table   points_nested_table_type)
NESTED TABLE
 points_nested_table STORE AS points_nested_tab;
```

The ENVIRONMENT_DATA table can now store an array of points. An array of (X,Y) points can represent the collection of data for a single sampling. The following PL/SQL block declares an object based on the newly created data dictionary types. The types POINTS_VARRAY_TYPE and POINTS_NESTED_TABLE_TYPE enable us to create database tables with columns of each type. Additionally, we can declare PL/SQL variables of these same types.

The first INSERT statement inserts a two-point array for the varray and nested table. Following this initial insert, the varray and nested table structures are populated with a series of points. The varray has five points; the nested table has four points. These point array structures are the second insert. Following that, the structures are read from the database with a SELECT statement. The key to this working is the types that have been created in the database; these same types are used for database columns and PL/SQL structures.

```
DECLARE
    a_points_varray points_varray_type
                    := points_varray_type();
    a_points_nested_table points_nested_table_type;
BEGIN
    INSERT INTO environment_data
        (sample_ID,  points_varray, points_nested_table)
    VALUES
        (1,
         points_varray_type
                 (point_type(3,4),point_type(3,5)),
         points_nested_table_type
                 (point_type(1,2),point_type(5,9))
        );

    a_points_varray := points_varray_type
                           (point_type(1,2),
                            point_type(2,3),
                            point_type(7,3),
                            point_type(2,8),
                            point_type(5,4));

    a_points_nested_table :=  points_nested_table_type
                           (point_type(1,2),
                            point_type(4,0),
                            point_type(7,3),
                            point_type(5,9));

    INSERT INTO environment_data
        (sample_ID, points_varray, points_nested_table)
    VALUES
        (2, a_points_varray, a_points_nested_table);

    SELECT  points_varray, points_nested_table
      INTO  a_points_varray, a_points_nested_table
      FROM  environment_data
     WHERE  sample_ID = 1;
END;
```

11.9.7 Objects

The material in this section is based on the nested table defined in the previous section.

You can build objects, with member functions, using types you create. We created the type POINTS_NESTED_TABLE TYPE. This can be used as a database table column type. It can be used as a PL/SQL variable type. We can create an object from this type. Such an object would be an array of points with member functions. A motivation for this would be the need to perform functions on the array points. For example, there is a constant need to examine the minimum X coordinate, maximum X coordinate, and average point. The following creates an object type. This is like the specification of the object. It declares member functions. The body is a second part.

The DDL to create this object is the following.

```
CREATE OR REPLACE TYPE points_object_type AS OBJECT
(
    points  points_nested_table_type,

    MEMBER FUNCTION sample_size RETURN NUMBER,
    MEMBER FUNCTION point_text RETURN VARCHAR2,
    MEMBER FUNCTION min_x RETURN NUMBER,
    MEMBER FUNCTION max_x RETURN NUMBER,
    MEMBER FUNCTION avg_x RETURN NUMBER,
    MEMBER FUNCTION best_point RETURN point_type,
    MEMBER procedure add_to(v point_type)
);
```

The object body is created with the following.

```
CREATE OR REPLACE TYPE body points_object_type AS
    MEMBER FUNCTION sample_size RETURN NUMBER IS
    BEGIN
        RETURN points.count;
    END sample_size;
    MEMBER FUNCTION point_text RETURN VARCHAR2
    IS
        s varchar2(1000);
    BEGIN
        FOR i IN 1..points.COUNT LOOP
            s := s || '('||points(i).x||','||points(i).x||')';
        END LOOP;
        RETURN s;
    END point_text;
    MEMBER FUNCTION min_x RETURN NUMBER
    IS
        result NUMBER := null;
    BEGIN
        FOR i IN 1..points.COUNT LOOP
            result :=
                least(nvl(result,points(i).x),points(i).x);
        END LOOP;
        return result;
    END min_x;
    MEMBER FUNCTION max_x RETURN NUMBER
    IS
        result NUMBER;
    BEGIN
        FOR i IN 1..points.COUNT LOOP
            result :=
                greatest(nvl(result,points(i).x),points(i).x);
        END LOOP;
        return result;
    END max_x;
    MEMBER FUNCTION avg_x RETURN NUMBER
    IS
        result NUMBER := 0;
    BEGIN
        FOR i IN 1..points.COUNT LOOP
            result := result + points(i).x;
        END LOOP;
        return (result/points.count);
    END avg_x;
    MEMBER FUNCTION best_point RETURN point_type
    IS
        pt point_type;
    BEGIN
        pt := point_type(points(1).x,points(points.COUNT).y);
        RETURN pt;
    END;
    MEMBER procedure add_to(v point_type) IS
    BEGIN
        points.EXTEND;
        points (points.count) := v;
    exception
        when others then points := points_nested_table_type(v);
    END add_to;
END;
```

The member functions are frequently short and simple. They only operate on a nested table structure consisting of a series of (X,Y) points. Building the member functions into the object can reduce the size and complexity of the application code. The following PL/SQL block illustrates a new ENVIRONMENT_DATA table. In this table, the column is not a nested table, but rather an object of type POINTS_OBJECT_TYPE.

```
CREATE TABLE environment_data
(
 sample_ID       NUMBER(3),
 points_object   points_object_type)
NESTED TABLE
 points_object.points STORE AS points_object_tab;
```

The following PL/SQL block declares an object of type POINTS_OBJECT_TYPE. Methods are used to populate and select from the object. The first method used is ADD_TO. This extends the nested table by adding a point. Four points are added to the object. An INSERT statement adds the object to the database table. The object is selected from the table, back into the original object declaration.

The calls to DBMS_OUTPUT print 2, 8, and 5.75 for the min, max, and average values respectively. The BEST_POINT method evaluates to the minumum X and maximum Y coordinate and prints X=2 Y=3.

```
DECLARE
    point_obj points_object_type :=
        points_object_type(points_nested_table_type());
    best_point point_type;
BEGIN
    point_obj.add_to(point_type(2,3));
    point_obj.add_to(point_type(6,1));
    point_obj.add_to(point_type(7,3));
    point_obj.add_to(point_type(8,3));

    INSERT INTO environment_data(sample_ID, points_object)
    VALUES
        (1, point_obj);

    SELECT points_object
    INTO   point_obj
    FROM   environment_data
    WHERE  sample_ID = 1;

    dbms_output.put_line(point_obj.min_x);
    dbms_output.put_line(point_obj.max_x);
    dbms_output.put_line(point_obj.avg_x);
    best_point := point_obj.best_point;

    dbms_output.put_line
        ('x='||best_point.x||' y='||best_point.y);
END;
```

11.9.8 Large Objects (LOBs)

Large objects refer to whole documents or images that need to be stored as column values in a table. Employee photographic images are examples of column values in an employee database. These binary graphic images are not manipulated but you can use PL/SQL to read images from one table and store into another. You can load images from host files into a database table. Binary images are stored as a BLOB in the database. The BLOB is also a PL/SQL type. Character objects are stored in the database as a CLOB. There is also a PL/SQL CLOB type. A BFILE is a database and PL/SQL type used to reference a file. All manipulation of large objects is through the Oracle DBMS_LOB package.

The following illustrates a scenario in which a table is used to store large character documents. The document is the DOC column. The ID column is a primary key used to identify a document.

The table to store documents is:

```
CREATE TABLE doc
(
     doc_id   NUMBER(5) PRIMARY KEY,
     document CLOB
);
```

We start with inserting an empty document using the Oracle built-in function EMPTY_CLOB. Later, the empty document can be referenced in PL/SQL. This will require declaring a CLOB type object that acts like a pointer to the empty document. To insert the empty document (using 1 for this DOC ID.)

```
INSERT INTO DOC (doc_id,document) VALUES (1, empty_clob());
COMMIT;
```

Consider the need to load host files into the DOC table. Assume the host files are large character documents and exist in a host directory called D:\DOCS.

We do not use host directory pathnames in the code. Oracle maintains logical to physical mappings in the view DBA_DIRECTORIES. You must create the logical connection with a CREATE DIRECTORY statement. After that, you code using the logical name. Create a directory in the database with the following.

```
CREATE OR REPLACE DIRECTORY SAMPLE_DOCS AS 'D:\DOCS';
```

To do this you need the CREATE ANY DIRECTORY privilege.

Assume you want to load this file into the DOC table:

```
users_manual.pdf
```

The following PL/SQL block uses the PL/SQL function BFILENAME to reference the file content. BFILENAME establishes the variable THE_BFILE as a reference to the file. After the BFILENAME call, the empty document is selected using the primary key of 1 from the preceding INSERT statement. Both the CLOB and BFILE are accessed in a manner similar to the way a pointer references memory. The content is not physically local to the procedure. It is referenced and accessed through the DBMS_LOB packages. The LOADFROMFILE copies the content. Then both references are closed. At completion, the host file content exists in the DOC table.

```
DECLARE
    the_bfile   BFILE;
    the_clob    CLOB;
    bfile_size  PLS_INTEGER;
    clob_size   PLS_INTEGER;
    v_directory VARCHAR2(30) := 'SAMPLE_DOCS';
    v_filename  VARCHAR2(30) := 'users_manual.pdf ';
BEGIN
    the_bfile := BFILENAME(v_directory, v_filename);

    SELECT document INTO the_clob FROM doc WHERE doc_id = 1
    FOR UPDATE OF doc.document NOWAIT;

    -- open the clob for update.
    dbms_lob.open(lob_loc => the_clob,
                  open_mode => dbms_lob.lob_readwrite);

    -- open the bfile
    dbms_lob.fileopen (file_loc => the_bfile,
                       open_mode => dbms_lob.file_readonly);

    -- what is the size of the bfile
    bfile_size := dbms_lob.getlength(the_bfile);

    -- load from file
    dbms_lob.loadfromfile (dest_lob => the_clob,
                           src_lob => the_bfile,
                           amount => bfile_size);

    -- check size after load.
    clob_size := dbms_lob.getlength(the_clob);

    -- close file
    dbms_lob.fileclose(file_loc => the_bfile);

    -- close blob
    dbms_lob.close(lob_loc => the_clob);

    commit;
END;
```

The following block extracts the CLOB object and prints it using DBMS_OUTPUT. Use a small file for this exercise. CLOBs are not intended to be printed using DBMS_OUTPUT. The Oracle REPLACE function, in the last line of this block, replaces a line feed with a dash. Otherwise, DBMS_OUTPUT prints the first line. This script reads the first 200 characters of the CLOB stored in the DOC table.

```
DECLARE
    the_clob        CLOB;
    clob_size       PLS_INTEGER;
    max_size        PLS_INTEGER := 200;
    amount_to_read  PLS_INTEGER;
    offSet          PLS_INTEGER:= 1;
    vbuf            VARCHAR2(200) := null;
BEGIN
    SELECT document INTO the_clob FROM doc WHERE doc_id=1;

    dbms_lob.open(the_clob, dbms_lob.lob_readonly);

    clob_size := dbms_lob.getlength(the_clob);

    amount_to_read := least(clob_size, max_size);

    dbms_lob.read(the_clob, amount_to_read, offset, vbuf);
    dbms_lob.close(the_clob);

    dbms_output.put_line(replace(vbuf,chr(10),'-'));
END;
```

This will print the first 200 characters of the CLOB, which you loaded from a BFILE, and which you first read with BFILENAME.

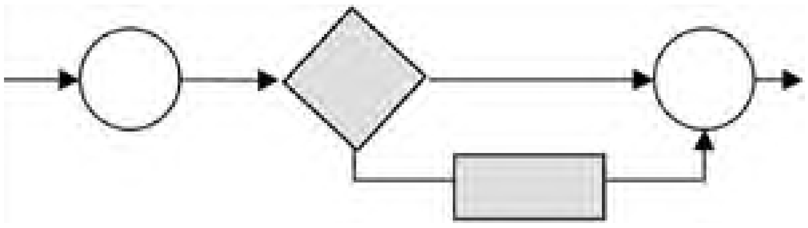
11.10 IF Statement

The IF construct implements IF-THEN-ELSE logic with optional ELSIF clauses.

11.10.1 Simple IF

The simple IF statement performs an action based on a true-false condition, illustrated by the diamond in Figure 11-1. If the condition evaluates TRUE, the code executes the logic in the box.

Figure 11-1. Simple IF.



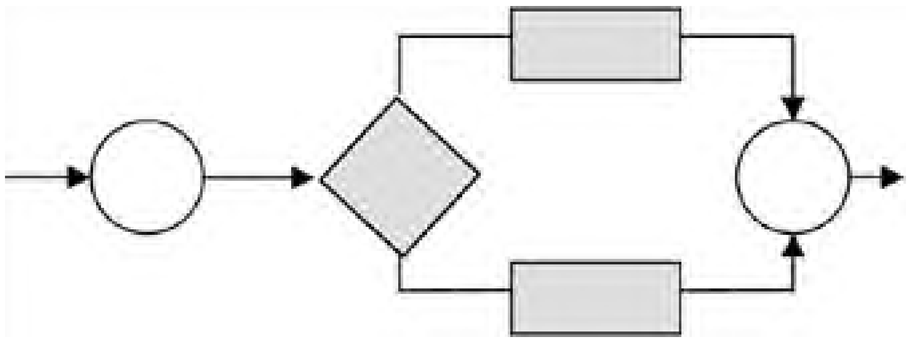
An example is:

```
IF (a < 10) THEN b := 1; END IF;
```

11.10.2 If-Then-Else

An IF-THEN-ELSE statement, shown in Figure 11-2, performs an action based on a true-false condition. An alternative action is always performed. If the test evaluates TRUE, the code executes the logic in that box, otherwise it executes the logic in the other box.

Figure 11-2. If-Then-Else.



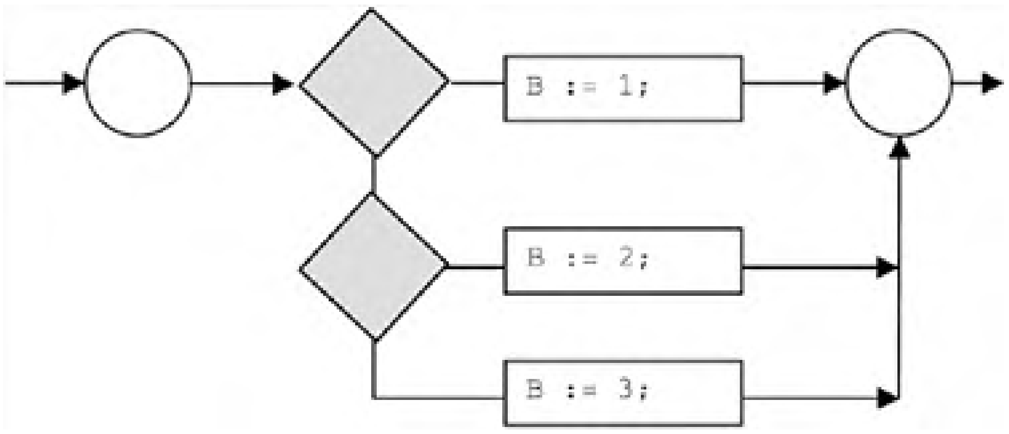
An example is:

```
IF (a < 10) THEN b := 1; ELSE b := 20; END IF;
```

11.10.3 If-Then-Elsif with Else

The IF-THEN-ELSIF-with-ELSE construct is shown in Figure 11-3. Any TRUE condition will execute code in that box and exit the construct. A FALSE condition enters the next decision. The logic in the ELSE part executes if no condition is TRUE.

Figure 11-3. If-Then-Elsif with Else.



In this example, B is set to 3 if no condition ever evaluates to TRUE

```
IF (a < 10) THEN
    b := 1;
ELSIF (a = 11) THEN
    b := 2;
ELSE
    b := 3;
END IF;
```

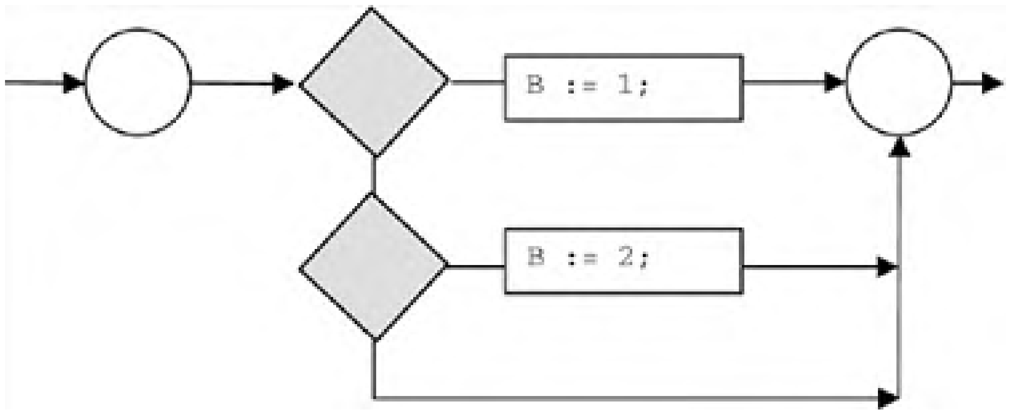
Multiple conditions may be true, but the logic of the first TRUE condition is executed, only. The following IF statement begins with assigning A the value of 8. The first test is TRUE, so is the second. After this IF statement, B is equal to 1.

```
a := 8;
IF (a < 10) THEN
    b := 1;
ELSIF (a < 20) THEN
    b := 2;
ELSE
    b := 3;
END IF;
```

11.10.4 If-Then-Elsif No Else

The IF-THEN-ELSIF-No-ELSE construct, shown in Figure 11-4, offers another option with IF statements. Any TRUE condition will execute code in that box and exit the construct. Each decision failure enters the next decision. If all tests evaluate to FALSE, no logic is executed. This is significantly different from IF statements that use an ELSE clause.

Figure 11-4. If-Then-Elsif No Else.



This example does not assign a value to B if the test conditions for A are each false.

```
IF (a < 10) THEN
    b := 1;
ELSIF (a = 11) THEN
    b := 2;
END IF;
```

11.10.5 Statement Expressions

The IF syntax includes an optional ELSIF and ELSE clause.

```
IF (multiline BOOLEAN expression) THEN
    Statement;
    Statement;
ELSIF (multiline BOOLEAN expression) THEN
    Statement;
    Statement;
ELSIF (multiline BOOLEAN expression) THEN
    Statement;
    Statement;
ELSE
    Statement;
    Statement;
END IF;
```

The following is a simple IF statement. This IF-THEN-ELSE statement assigns a value to B for all values of A. An IF statement can be coded in such a manner that it overlooks some conditions. Sometimes this is intended. Sometimes it is a bug.

```
IF (a < 10) THEN B := 1; END IF;
```

The previous IF statement only considers the case where A is less than 10. This next statement takes an action on B for any value of A.

```
IF (a < 10) THEN b := 1; ELSE b := 2; END IF;
```

An IF statement can use the PL/SQL NULL statement to indicate that all conditions are being considered. For example, the following IF-THEN-ELSE contains a PL/SQL NULL statement to explicitly state that no action is to be taken if A is greater than 10.

```
IF (a <= 10) THEN
    b := 1;
ELSE
    NULL;
END IF;
```

An IF-THEN-ELSIF statement does not have to include an ELSE clause. The following takes no action if A is greater than 10 and not equal to 20.

```
IF (a < 10) THEN
    b := 1;
ELSIF (a = 20) THEN
    b := 2;
END IF;
```

For this next statement, if A is less than 10, B is assigned 1, not 2. When an IF condition is met, no other conditions are considered.

```
IF (a < 10) THEN
    b := 1;
ELSIF (a <= 20) THEN
    b := 2;
ELSE
    b := 3;
END IF;
```

Decision tables are useful tools for planning lengthy IF statements. A decision table is two columns with the left column condition determining the value of the right column. A decision table can quickly identify a missing condition. A sample decision table for changing values of B based on A is the following:

A	B
A < 10	1
10 >= A < 20	2
225	3
All other values	4

The left column of the decision table includes all values in the range of A. The statement for this is the following.

```
IF (A < 10) THEN
   B := 1;
ELSIF (A  >= 10 AND A < 20) THEN
   B := 2;
ELSIF (A = 225) THEN
   B := 3;
ELSE
   B := 4;
END IF;
```

The IF condition can be any statement that evaluates to TRUE or FALSE. You have a package called BUSINESS_DAYS and it contains a function that returns the next business date. Showing this package and the function, NEXT_BUSINESS_DAY:

```
PACKAGE business_days IS
   function next_business_day RETURN DATE;
   other functions, etc
END business_days;
```

You can use the logic in this package within an IF condition. Assume your code has a DATE variable called TRANSACTION_DATE. Use the logic built into the existing NEXT_BUSINESS_DATE function with the following code:

```
IF (transaction_date = business_days.next_business_day) THEN
```

It is common to have packages that are developed to encapsulate the logic of business rules. Should the rule for the next business day change, only the BUSINESS_DAY package is updated. The code that uses the package is not changed.

11.10.6 Use DECODE and CASE in SQL

Unnecessary logic can easily creep into PL/SQL procedures. Consider the transformations needed when querying data from the database. SQL CASE expressions have built-in IF-THEN_ELSE logic. The DECODE statement performs value transformations. Both are illustrated with examples.

Consider the following IF statement that assigns a value to B based on the value in A—logic of this kind can and should be minimized where possible with SQL functions.

```
IF (A < 10) THEN
   B := 1;
ELSIF (A  >= 10 AND A < 20) THEN
   B := 2;
ELSIF (A = 225) THEN
   B := 3;
ELSE
   B := 4;
END IF;
```

Reduce and eliminate the type of logic shown by performing the transformation within the SQL query. The following illustrates how a SQL CASE statement performs the equivalent transformations used in the previous IF statements.

```
CREATE TABLE TEMP (A NUMBER(2));
INSERT INTO TEMP VALUES (11);

DECLARE
   B INTEGER;
BEGIN
   SELECT CASE
         WHEN A < 10             THEN 1
         WHEN A >= 10 AND A < 20 THEN 2
         WHEN A = 225            THEN 3 END
   INTO B FROM TEMP;
   dbms_output.put_line('B='||b);
END;
```

SQL statements with DECODE transform values when the data is selected from the database. The following SQL queries the college major descriptions from the MAJOR_LOOKUP table. Within the query each major description is transformed to an alternative string.

```
SELECT DECODE
      ( major_desc,
       'Undeclared'   , 'A1',
       'Biology'      , 'A2',
       'Math/Science' , 'A3',
       'History'      , 'A4',
       'English'      , 'A5') major
FROM major_lookup;
```

The SQL CASE expression is more powerful that DECODE. Use CASE and DECODE to initially transform data. This simplifies the PL/SQL logic.

11.11 CASE Statement

Case statements implement the same construct model as IF-THEN-ELSIF with ELSE constructs. The CASE statement is easier to read and has improved performance over complex IF statements. There are two forms to the CASE statement:

- Searched Case Statement
- Case with Selector

11.11.1 Searched CASE Statement

The searched CASE statement evaluates a sequence of test conditions. When a test evaluates to TRUE, code is executed and the construct is complete—no further conditions are examined.

Execution falls to the ELSE clause if no prior conditions are met. The ELSE clause can be any group of statements or the NULL statement. NULL statements refer to the PL/SQL statement, NULL.

If no condition is met and there is no ELSE clause, an exception is raised. The error is:

```
ORA-06592: CASE not found while executing CASE statement.
```

This error can be captured with the exception:

```
CASE_NOT_FOUND
```

This is illustrated later. The Searched Case Statement evaluates an expression with each WHEN clause. The syntax is the following:

```
CASE
   WHEN expression THEN action;
   WHEN expression THEN action;
   WHEN expression THEN action;
   [ELSE action;]
END CASE;
```

The following example defines a rule for setting a value to B based on the value of A. We implement this rule with a CASE statement.

A	B
A < 10	1
10 >= A < 20	2
225	3
All other values	4

The CASE statement for this is:

```
CASE
   WHEN (A < 10)              THEN B := 1;
   WHEN (A >= 10 AND A < 20)  THEN B := 2;
   WHEN (A = 225)             THEN B := 3;
   ELSE                            B := 4;
END CASE;
```

The ELSE clause is optional. The following CASE statement is valid; however, this code raises an exception because no condition is TRUE. The exception handler catches the error and prints the value A.

```
DECLARE
    A INTEGER := 300;
BEGIN
    CASE
        WHEN (A < 10)              THEN B := 1;
        WHEN (A >= 10 AND A < 20)  THEN B := 2;
        WHEN (A = 225)             THEN B := 3;
    END CASE;
EXCEPTION
    WHEN CASE_NOT_FOUND THEN
        dbms_output.put_line('A = '||a);
END;
```

The CASE statement is easier to read in the code. Compare the following CASE logic with the same IF logic.

Search Case Statement	IF Statement

CASE
WHEN (A<=100) THEN B := 1;
WHEN (A<=200) THEN B := 2;
WHEN (A<=300) THEN B := 3;
ELSE   B := 4;
END CASE;
	IF (A <= 100) THEN
	B := 1;
	ELSIF (A <= 200) THEN
	B := 2;
	ELSIF (A <= 300) THEN
	B := 3;
	ELSE
	B := 4;
	END IF;

11.11.2 CASE with Selector

The CASE with Selector also raises an exception if no conditions are true and there is no ELSE clause. The syntax for this CASE statement is:

```
CASE selector
    WHEN value THEN action;
    WHEN value THEN action;
    WHEN value THEN action;
    [ELSE action;]
END CASE;
```

The following example prints a two-character string for each college major. An exception handler is not necessary because there is an ELSE clause.

```
DECLARE
    college_major major_lookup.major_desc%TYPE;

    PROCEDURE p(s VARCHAR2) IS
    BEGIN
        dbms_output.put_line(s);
    END;

BEGIN
    college_major := 'Biology';

    CASE college_major
        WHEN 'Undeclared'   THEN p('A1');
        WHEN 'Biology'      THEN p('A2');
        WHEN 'Math/Science' THEN p('A3');
        WHEN 'History'      THEN p('A4');
        WHEN 'English'      THEN p('A5');
        ELSE                     p('none');
    END CASE;
END;
```

11.11.3 Using CASE within the SELECT

Use SQL CASE expressions in SQL query statements first. A more lengthy procedure will use a basic SQL SELECT statement that immediately transforms the data using a Searched Case Statement. The following illustrates the difference.

We create a TEMP table. The code under Version 1 executes a SELECT statement and then transforms the data. Version 2 uses CASE within the SELECT statement. This data transform occurs in the SQL engine. The logic of a procedure following Version 2 will be simpler.

```
CREATE TABLE TEMP (A NUMBER(2));
INSERT INTO TEMP VALUES (11);
```

VERSION 1

```
DECLARE
    A INTEGER;
    B INTEGER;
BEGIN
    SELECT A INTO A FROM TEMP;
    CASE
        WHEN (A<=100)  THEN B := 1;
        WHEN (A<=200)  THEN B := 2;
        WHEN (A<=300)  THEN B := 3;
        ELSE                B := 4;
    END CASE;
    dbms_output.put_line(B);
END;
```

VERSION 2

```
DECLARE
    B INTEGER;
BEGIN
    SELECT CASE
            WHEN A < 10             THEN 1
            WHEN A >= 10 AND A < 20 THEN 2
            WHEN A = 225            THEN 3 END
    INTO B FROM TEMP;
END;
```

For the Version 1 and Version 2 examples, the total lines of code are not that different; however, for larger applications, procedures following the style in Version 2 will be easier to read and maintain.

11.11.4 Using DECODE within the SELECT

Use SQL DECODE expressions in the SELECT, rather than a SQL SELECT statement, that immediately transforms the data using a Searched With Selector.

The following illustrates the difference. The code in Version 1 selects a college major and then uses CASE to transform the major into a two-character string. This transformation occurs in the PL/SQL. The Version 2 code performs the transformation in the query using DECODE.

VERSION 1

```
DECLARE
    college_major major_lookup.major_desc%TYPE;
    code VARCHAr2(4);
BEGIN
    SELECT major_desc INTO college_major
    FROM major_lookup WHERE ROWNUM = 1;

    -- for example, college_major := 'Biology';

    CASE college_major
        WHEN 'Undeclared'   THEN code := 'A1'
        WHEN 'Biology'      THEN code := 'A2');
        WHEN 'Math/Science' THEN code := 'A3');
        WHEN 'History'      THEN code := 'A4');
        WHEN 'English'      THEN code := 'A5');
        ELSE                     code := 'none';
    END CASE;
END;
```

VERSION 2

```
DECLARE
    code VARCHAR2(4);
BEGIN
    SELECT DECODE
       ( major_desc,
        'Undeclared'   , 'A1',
        'Biology'      , 'A2',
        'Math/Science' , 'A3',
        'History'      , 'A4',
        'English'      , 'A5') major
    FROM major_lookup WHERE ROWNUM = 1;
END;
```

11.12 LOOP

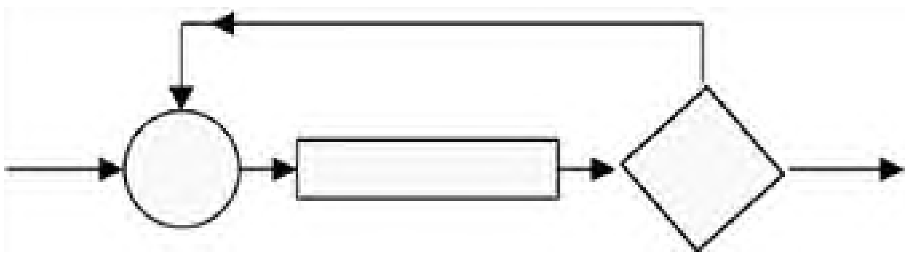
PL/SQL supports the following loop constructs:

- Infinite LOOP with an exit condition
- FOR LOOP
- WHILE LOOP

11.12.1 DO UNTIL LOOP

The DO UNTIL LOOP always performs some piece of work, until an exit condition occurs. This LOOP can terminate with an EXIT WHEN statement or an IF condition with an EXIT. If no termination condition occurs then this becomes an infinite loop. As with any loop, an exception, such as divide by zero, can break the loop. Figure 11-5 shows the DO-UNTIL LOOP construct.

Figure 11-5. DO UNTIL LOOP.



The DO UNTIL LOOP always makes at least one pass through the loop block. Following each pass is the test for continuing. The syntax templates are:

- A loop that exits with an IF statement:

```
LOOP
  <statements>
  IF (expression) THEN EXIT; END IF;
END LOOP;
```

- A loop that exits with an EXIT WHEN statement:

```
LOOP
  <statements>
  EXIT WHEN <boolean expression>
END LOOP;
```

This decrements a count and uses EXIT WHEN to terminate the loop at zero.

```
DECLARE
  counter INTEGER := 10;
BEGIN
  LOOP
    dbms_output.put_line(counter);
    counter := counter - 1;
    EXIT WHEN counter = 0;
  END LOOP;
END;
```

The same functionality is implemented with an IF condition. The difference between the previous block and the following is style.

```
DECLARE
    counter INTEGER := 10;
BEGIN
    LOOP
        dbms_output.put_line(counter);
        counter := counter - 1;
        IF (counter = 0) THEN EXIT; END IF;
    END LOOP;
END;
```

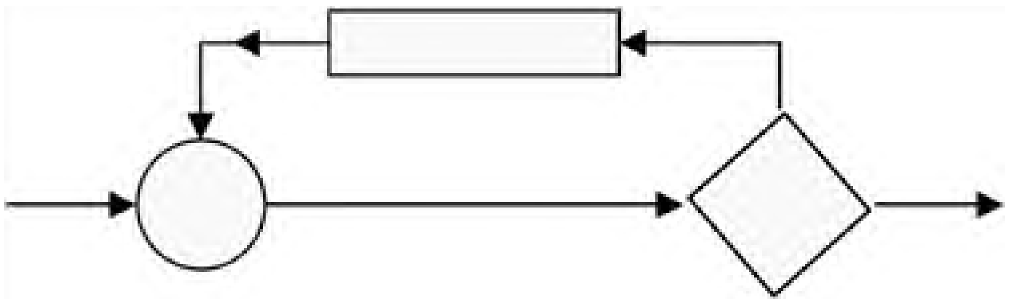
The following PL/SQL block decrements a variable that has a POSITIVE datatype. POSITIVE type variables must be greater than zero. When COUNTER is decremented to zero, the exception is raised. The exception handler catches the exception. This relies on neither an EXIT nor an IF condition to terminate the loop.

```
DECLARE
    counter POSITIVE := 10;
BEGIN
    LOOP
        BEGIN
            dbms_output.put_line(counter);
            counter := counter - 1;
        EXCEPTION WHEN VALUE_ERROR THEN EXIT;
        END;
    END LOOP;
END;
```

11.12.2 WHILE LOOP

The WHILE LOOP tests for a condition prior to any logic execution. The number of passes through the logic may be zero. The WHILE LOOP construct is shown in Figure 11-6.

Figure 11-6. WHILE LOOP.



The syntax for this construct is

```
WHILE (condition is true) LOOP
    <statements>
END LOOP;
```

The condition test must be a Boolean expression that evaluates to TRUE or FALSE. The following loop generates a lower case alphabet list.

```
DECLARE
    ascii_char INTEGER := 97;
    alphabet   VARCHAR2(26);
BEGIN
    WHILE
        (NVL(LENGTH(alphabet),0) < 26)
    LOOP
        alphabet := alphabet || chr(ascii_char);
        ascii_char := ascii_char + 1;
    END LOOP;
    dbms_output.put_line(alphabet);
END;
```

Evaluation of the WHILE expression should not include a NULL because NULL cannot evaluate to TRUE or FALSE. A NULL length for a string is used to illustrate this. The length of a string, which has no assignment, has no length. That is, if we declare a string as follows:

```
alphabet VARCHAR2(26);
```

The length of that string, prior is any assignment, is null. The following expression does not return 0 or a number, but NULL.

```
LENGTH(alphabet)
```

Had the PL/SQL WHILE LOOP been written as follows, the LOOP would never make a single pass because the WHILE condition would never evaluate to TRUE.

```
WHILE
   (LENGTH(alphabet) < 26)
LOOP
   Loop logic
END LOOP;
```

11.12.3 FOR LOOP

The FOR LOOP is a form of WHILE LOOP and has the following syntax.

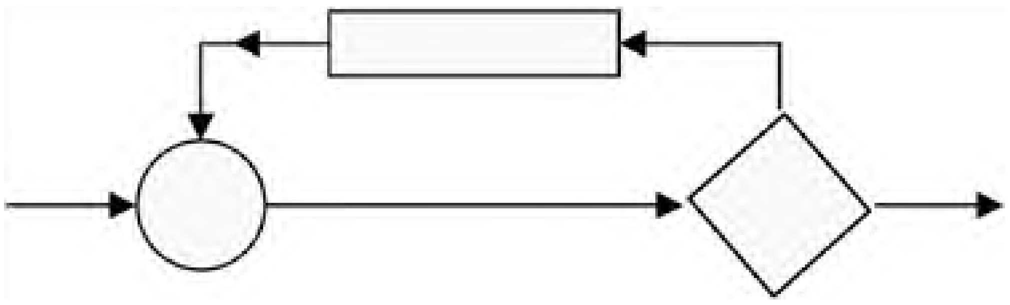
```
FOR C IN [REVERSE] A . . B LOOP
   <statements>
END LOOP;
```

From a first appearance, it seems that a FOR LOOP always makes at least one pass. This is not always the case. The symbols A and B in the preceding syntax can be literals or expressions that evaluate to an integer value. If, upon entry into the loop, A is greater than B, then the statements in the loop will not execute. In summary:

	Upon initial loop entry
A < B	Statements execute several times depending on values for A and B.
A = B	Statements execute ONCE—only one pass through the loop.
A > B	The loop is skipped—no iteration of the loop.

The loop construct for the FOR LOOP always includes an initial evaluation of (B-A), which will exit when this is less than C. Figure 11-7 shows the FOR LOOP construct.

Figure 11-7. FOR LOOP.



The following block contains a FOR LOOP where, upon initial entry, (B-A) is negative. Therefore there will be no DBMS_OUTPUT from this block except the message that the code never entered the loop:

```
DECLARE
    str VARCHAR2(10);
    x   integer := 11;
    y   integer := 10;
BEGIN
    FOR i in x..y LOOP
        str := str || 'abc';
    END LOOP;
    dbms_output.put_line(NVL(str, 'Never entered the loop.'));
END;
```

The FOR LOOP counter is shown in the syntax as the letter C.

```
FOR C IN [REVERSE] A . . B LOOP
    <statements>
END LOOP;
```

This counter is an implicitly declared variable. You choose the name of this variable. You can duplicate the name of another variable—this will make the code confusing to read, but it will work. The following declares a variable and a loop counter by the same name, which is confusing to the reader.

```
DECLARE
   my_counter INTEGER := 100;
BEGIN
    FOR my_counter in 1..2 LOOP
        dbms_output.put_line(my_counter);
    END LOOP;
    dbms_output.put_line(my_counter);
END;
```

The loop counter, MY_COUNTER, has scope only inside the loop. Hence the DBMS_OUTPUT from within the loop will print 1 and 2. The PL/SQL block variable MY_COUNTER has scope over the entire block. The last DBMS_OUTPUT prints 100.

To keep code simple, use simple, nonduplicating names for loop counters. Loop counters only have scope within the loop. The following will not compile because MY_COUNTER is referenced out of scope.

```
BEGIN  -- THIS WILL NOT COMPILE
    FOR my_counter in 1..2 LOOP
        dbms_output.put_line(my_counter);
    END LOOP;
    dbms_output.put_line(my_counter);
END;
```

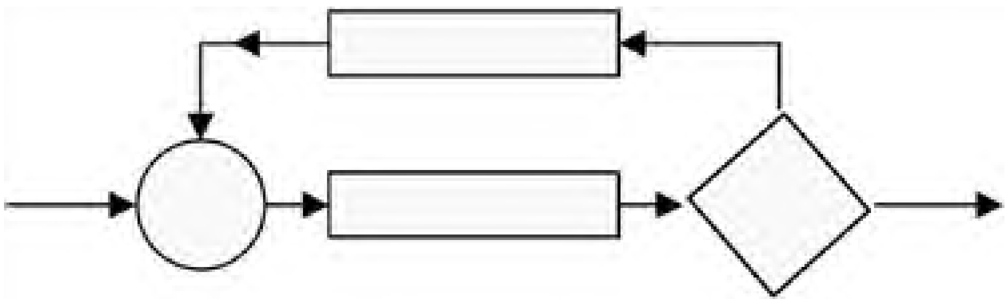
You can always capture the counter by copying it into another variable. The counter is implicitly declared, but it can be referenced like any IN mode variable. You can read it but you cannot change it. The PL/SQL with a counter to the left of an assignment operator will not compile. The following block captures the counter and prints the number of iterations made during the loop.

```
DECLARE
    number_of_passes INTEGER;
BEGIN
    FOR ctx in 1..2 LOOP
        number_of_passes := ctx;
    END LOOP;
    dbms_output.put_line('counter='||number_of_passes);
END;
```

11.12.4 DO-WHILE-DO LOOP

The DO-WHILE-DO LOOP construct, shown in Figure 11-8, is very common in applications. In this construct the code performs some task and then tests a condition. It continues with a second task if the test is successful. A common scenario is the use of an explicit cursor. This loop fetches a row and tests for an end-of-cursor. The second task processes the fetched row.

Figure 11-8. DO-WHILE-DO LOOP.



The termination of the loop can be with an EXIT WHEN of IF THEN EXIT statement. The syntax for this construct is the following. The first loop uses an IF THEN EXIT to terminate the loop.

```
LOOP
    <statements>
    IF (expression) THEN EXIT; END IF;
    <statements>
END LOOP;
```

This loop terminates with an EXIT WHEN.

```
LOOP
    <statements>
    EXIT WHEN <boolean expression>
    <statements>
EXIT LOOP;
```

The following is an explicit cursor loop that fetches student names from the STUDENTS table. The first task is to fetch the row. The EXIT condition is based on the success or failure of fetching another row. The second task in the loop is to print the student name.

```
DECLARE
    CURSOR C1 IS
        SELECT student_name
        FROM   students;
    cursor_record C1%ROWTYPE;
BEGIN
    OPEN C1;
    LOOP
        FETCH C1 INTO cursor_record;
        EXIT WHEN C1%NOTFOUND;
        dbms_output.put_line(cursor_record.student_name);
    END LOOP;
END;
```

11.12.5 Encapsulating the Logic of the Loop

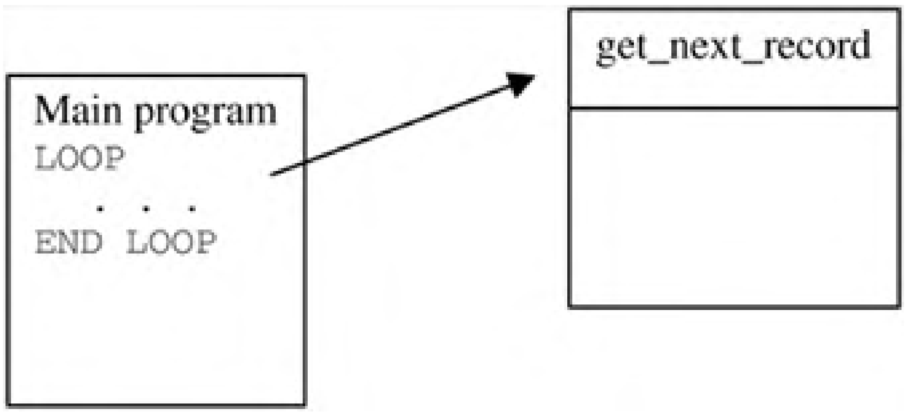
The logic within a loop can be extensive and difficult to read, especially if it spans pages. Portions of the logic can be broken out into separate procedures; these procedures often form the basis for new application packages. By partitioning the code into small modules, the loop is short and simple. The loop body contains a few procedure and function calls. The following is a template for a loop that must read and process records from a file.

```
LOOP
    get next record          -- this would be a procedure call.
    if end of file then exit –- uses a function.
    process record           -- this would be a procedure call.
END LOOP
```

This loop can be written with all the detail logic coded directly in the loop. This could result in a loop that spans a page or two. An alternative approach is to encapsulate major components of the loop body into specific procedures and functions. The loop body calls these subprograms. The following illustrates this strategy.

For this application, a building block will be a subprogram that simply returns a record from a file. Once we write that subprogram, we can use it in the loop body. This is illustrated in Figure 11-9.

Figure 11-9. Get Next Record.



The function to fetch a record must have a FILE argument as an IN parameter and a VARCHAR2 as an out parameter. The GET_NEXT_RECORD is designed to encapsulate any exception handling and return a TRUE/FALSE indicator for a successful read. The code for GET_NEXT_RECORD is.

```
CREATE OR REPLACE FUNCTION get_next_record
    (FILE IN  utl_file.file_type,
     text OUT VARCHAR2) RETURN BOOLEAN IS
BEGIN
    utl_file.get_line(FILE, text);
    RETURN false;
EXCEPTION
        WHEN no_data_found THEN return true;
END get_next_record;
```

The application loop continues processing based on a successful or unsuccessful read. There is no exception handling in the loop. The exception is encapsulated in the function GET_NEXT_RECORD. This is a DO-WHILE-DO loop. The core part of the application code is the following.

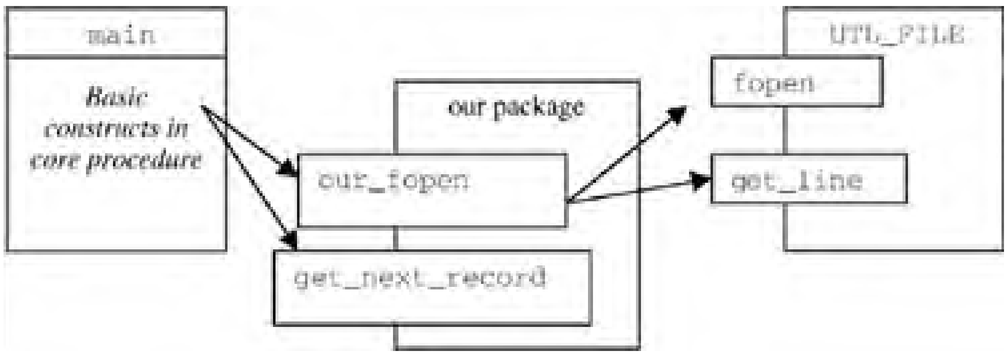
```
LOOP
    end_of_file := get_next_record(FILE, text);
    exit when end_of_file;
    dbms_output.put_line(text);
END LOOP;
```

The core of an application program can be easily muddled with the exceptions exported by application programs such as UTL_FILE. The function GET_NEXT_RECORD can form a basis for encapsulating an interface to a package such as UTL_FILE. This layered interface allows applications to write their core logic based on simple constructs such as loops controlled by conditional expressions. A complete PL/SQL block that uses the GET_NEXT_RECORD function but also contains the UTL_FILE call to open the file is the following.

```
DECLARE
    FILE utl_file.file_type;
    text VARCHAR2(120);
    end_of_file BOOLEAN := FALSE;
BEGIN
    FILE := utl_file.fopen('C:/TEST,'test.txt','r');
    LOOP
        end_of_file := get_next_record(FILE, text);
        exit when end_of_file;
        dbms_output.put_line(text);
    END LOOP;
    utl_file.fclose(FILE);
EXCEPTION
    WHEN UTL_FILE.INVALID_PATH THEN
        dbms_output.put_line('a');
    WHEN UTL_FILE.INVALID_MODE THEN
        dbms_output.put_line('b');
    WHEN UTL_FILE.INVALID_OPERATION THEN
        dbms_output.put_line('c');
END;
```

A more desirable application for the aforementioned PL/SQL block would be the following architecture in Figure 11-10.

Figure 11-10. Package with Get Next Record.



Following the model in Figure 11-10 the application loop acquires a style as shown next.

```
DECLARE
   token our_package.token_type;
   EOF   BOOLEAN;
   str   VARCHAR2(200);
LOOP
   end_of_file
      := our_package.get_next_record(token, str);
   exit when end_of_file;
   process file record returned.
END LOOP;
```

11.13 STRING Manipulation Functions

11.13.1 SUBSTR

```
SUBSTR(str, start_char, [no_of_chars])
```

str	varchar2	The input string.
start_pos	integer	Go to the START_POS character position in the string and count forward, NO_OF_CHARS. If START_POS is negative, start from the tail of the string.
no_of_chars	integer	Number of characters to select. If not specified, then get the rest of the string.
retuns	varchar2	Returns the substring or NULL.

The SUBSTR function returns a slice of an input string. The arguments START_POS and NO_OF_CHARS can be literals, variables, or string expressions that evaluate to a number. The following are some SUBSTR examples in a SQL*Plus session.

Given the string, 12345, select the tail of the string starting at character position 2.

```
SQL> SELECT substr('12345',2) str FROM dual;

STR
----
2345
```

Given the string 12345678, go to the fourth character from the end—make that the starting point and select the next two characters.

```
SQL> SELECT substr('12345678',-4,2) str FROM dual;

STR
----
56
```

Concatenate the last character and first character, in reverse. The first SUBSTR uses START_POS of negative 1—start counting from the right, just one character. This is concatenated with the string's first character.

```
SQL> SELECT substr('12345',-1)||substr('12345',1,1) str
FROM dual;

STR
---
51
```

START_POS and NO_OF_CHARS can be an expression. This example uses the INSTR function for the START_POS value. The objective is to select the file name portion from a full pathname. First, use INSTR to return the last position of a forward slash in a string. In this example, the position of the last forward slash is the 10th character position.

```
SQL> SELECT instr('/aa/bb/cc/dd','/',-1,1)  FROM dual;

INSTR('/AA/BB/CC/DD','/',-1,1)
------------------------------
                            10
```

To select just the filename from a full pathname, select all remaining characters after the last forward slash—for the previous select, that would be all characters after the 10th position. This can be generalized to the evaluation of: the INSTR function result plus 1.

Incorporate "INSTR result + 1" into a PL/SQL function. This produces a function that can return the filename portion of a pathname.

```
FUNCTION filename(v_path IN VARCHAR2) RETURN VARCHAR2
IS
BEGIN
    RETURN substr(v_path, instr(v_path,'/',-1,1)+1);
END filename;
```

Users of this function can extract the filename portion of a string by coding:

```
my_file_name := filename(v_full_pathname);
my_file_name := filename('dir/dir/filename.dbf');
```

If the value passed, V_FULL_PATHNAME does not contain a forward slash (a caller passed a filename only) and INSTR returns a 0. To this 0 is added a 1. This evaluates to the following:

```
RETURN substr(v_path, instr(v_path,'/',-1,1)+1);
RETURN substr(v_path, 0+1);
```

The function RETURN value is the original string, which would be the file name.

SUBSTR returns NULL if the START_POS exceeds the length of a string. This can occur if you code a SUBSTR function with a literal START_POS value. The following example uses 10. If the argument passed to this procedure is less than 10 characters, the SUBSTR function result is NULL.

```
PROCEDURE test (arg IN VARCHAR2)
IS
    declarations
BEGIN
    my_variable := SUBSTR(arg, 10);
    remaining PL/SQL
END
```

The procedure TEST may not produce the expected results if ARG is less than 10 characters. Avoid situations like this by using NVL and not using literals. You can always check the length of a string with the LENGTH function.

11.13.2 INSTR

```
INSTR(str1, str2, [start_pos [,occurrence]])
```

Str1	varchar2	STR1 is the string being searched.
Str2	varchar2	We are looking for an occurrence of this string within STR1.
start_pos	integer	Start looking at this character position.
occurrance	integer	Look for the first, second, or nth occurrence.
returns	integer	Returns the character position or 0.

The INSTR function returns a number, which is the character position of the nth occurrence of STR2 within STR1.

Zero is returned if no substring or the requested occurrence of that string cannot be found. To experiment with a comma-delimited string, hard code a string using DUAL. This SQL returns the first occurrence of a comma.

```
SQL> SELECT instr('aaa,bbb,ccc,ddd',',') FROM dual;

INSTR('AAA,BBB,CCC,DDD',',')
----------------------------
                           4
```

The second occurrence of a comma, starting from character position 3, is the 8th character.

```
SQL> select instr('aaa,bbb,ccc,ddd',',',3,2) from dual;

INSTR('AAA,BBB,CCC,DDD',',',3,2)
--------------------------------
                               8
```

The third occurrence of a comma, starting from character position 2, is the 12th character.

```
SQL> select instr('aaa,bbb,ccc,ddd',',',2,3) from dual;

INSTR('AAA,BBB,CCC,DDD',',',2,3)
--------------------------------
                              12
```

Encapsulate INSTR into a function. The function NEXT_TOKEN is a building block for code that parses delimited strings. This function returns TRUE if the substring is found. An OUT mode parameter, FOUND_AT_POSITION, identifies the location of the substring.

```
CREATE OR REPLACE FUNCTION next_token
    (base_string       IN VARCHAR2,
     looking_for       IN VARCHAR2,
     start_looking_at  IN INTEGER,
     found_at_position OUT INTEGER) RETURN BOOLEAN
IS
    result integer;
BEGIN
    found_at_position :=
        INSTR (base_string,looking_for,
               start_looking_at,1);
    RETURN (found_at_position <> 0);
END next_token;
```

Each call to NEXT_TOKEN returns the FOUND_AT_POSITION. Hence repeated calls, each passing FOUND_AT_POSITION, will return all occurrences of the token. For example, pass a comma-delimited string and display each FOUND_AT_POSITION. The following is a sequence of calls to NEXT_TOKEN. Each subsequent call begins the search with FOUND_AT_POSITION + 1.

```
DECLARE
   str         VARCHAR2(100) := 'aaa,bbb,ccc,ddd,eee,fff';
   delimeter   VARCHAR2(1) := ',';
   b           BOOLEAN;
   found_at    INTEGER;
   start_point INTEGER := 1;
BEGIN
    b := next_token(str,delimeter,start_point,found_at);
    dbms_output.put_line(found_at);

    start_point := found_at + 1;
    b := next_token(str,delimeter,start_point,found_at);
    dbms_output.put_line(found_at);

    start_point := found_at + 1;
    b := next_token(str,delimeter,start_point,found_at);
    dbms_output.put_line(found_at);
END;
```

The result of this PL/SQL block is the location of the first three commas.

```
4
8
12
```

Rather than code a repeated sequence, use a WHILE LOOP that parses the entire string. The loop continues to cycle provided the return code is TRUE.

```
DECLARE
   str         VARCHAR2(100) := 'aaa,bbb,ccc,ddd,eee,fff';
   delimeter   VARCHAR2(1) := ',';
   b           BOOLEAN;
   found_at    INTEGER;
   start_point INTEGER := 1;
BEGIN
    WHILE
        (next_token(str,delimeter,start_point,found_at))
    LOOP
        dbms_output.put_line(found_at);
        start_point := found_at + 1;
    END LOOP;
END;
```

The WHILE LOOP produces all comma occurrences: 4, 8, 12, 16, and 20. Using this data we can select all fields between the commas. The first field is in character positions 1–3, the second in positions 5–7. These are start and stop points of the string. We can use SUBSTR to get these fields, but SUBSTR requires a starting point and a length. We can write an enhancement to SUBSTR. This will be a new function that uses start and stop points. This function is written as SUB_STRING.

```
CREATE OR REPLACE FUNCTION sub_string
    (base_string IN VARCHAR2,
     start_point IN INTEGER,
     end_point IN INTEGER) RETURN VARCHAR2 IS
BEGIN
    RETURN SUBSTR(
        base_string,
        start_point,
        end_point-start_point+1);
END sub_string;
```

There are currently two string functions: NEXT_TOKEN and SUB_STRING. These functions are likely candidates for a string manipulation package, called STRINGS_PKG. The package specification is shown here.

```
CREATE OR REPLACE PACKAGE strings_pkg IS
FUNCTION next_token
    (base_string       IN  VARCHAR2,
     looking_for       IN  VARCHAR2,
     start_looking_at  IN  INTEGER,
     found_at_position OUT INTEGER) RETURN BOOLEAN;
    FUNCTION sub_string
        (base_string      IN  VARCHAR2,
         start_point      IN  INTEGER,
         end_point        IN  INTEGER) RETURN VARCHAR2;
END strings_pkg;
```

There is no need to show the package body. That will consist only of the procedure code for NEXT_TOKEN and SUB_STRING, previously shown. When the final application code is written using the strings package, it has the following form.

```
DECLARE
   str         VARCHAR2(100) := 'aaa,bbb,ccc,ddd,eee,fff';
   delimeter   VARCHAR2(1) := ',';
   b           BOOLEAN;
   found_at    INTEGER;
   start_point INTEGER := 1;
BEGIN
    WHILE
      (strings_pkg.next_token
          (str,delimeter,start_point,found_at))
    LOOP
        dbms_output.put_line
            (strings_pkg.sub_string
                (str,start_point,found_at-1));
        start_point := found_at + 1;
    END LOOP;

    dbms_output.put_line(substr(str,start_point));
END;
```

The output from this application piece now identifies each token in the string.

```
aaa
bbb
. . .
fff
```

11.13.3 LPAD, RPAD

```
LPAD(str1, new_length [,str2])
RPAD(str1, new_length [,str2])
```

str1	varchar2	The string to be padded.
new_length	integer	The new length of the string. Pad the string to make it this long.
str2	integer	The pad—append this to STR1. If no STR2 is passed, a BLANK character is used for padding.
returns	varchar2	The padded string that may be padded with STR2, or actually truncated—this depends on the length of STR1 and new_length.

The PAD function can be used to pad characters such as a blank or a zero. The pad can be a string—not just a character. See complementary functions that trim: LTRIM, RTRIM.

The following table contains five SQL statements. The first three statements left-pad a three-character string. The result is a six-character string. The fourth example does no padding because the string is already six characters. The last SQL statement truncates the string—STR1 exceeds NEW_LENGTH, which is 6, and is truncated.

The SQL statement	Returns
SQL> SELECT LPAD('abc',6,'0') FROM dual;	000abc
SQL> SELECT LPAD('abc',6,'01') FROM dual;	010abc
SQL> SELECT LPAD('abc',6,'01234') FROM dual;	012abc
SQL> SELECT LPAD('abcdef',6,'01234') FROM dual;	abcdef
SQL> SELECT LPAD('abcdefg',6,'01234') FROM dual;	abcdef

The LPAD function truncates a string when NEW_LENGTH exceeds the length of the string. A string function can be built with some defensive code that takes this into consideration. The following code replaces NEW_LENGTH with the greater of LENGTH(STR1) and NEW_LENGTH—this prevents string truncation.

```
CREATE OR REPLACE FUNCTION left_pad
    (base_string  IN VARCHAR2,
     new_length   IN INTEGER,
     pad_string   IN VARCHAR2) RETURN VARCHAR2
IS
BEGIN
    RETURN LPAD
        (base_string,
         GREATEST(LENGTH(base_string), new_length),
         pad_string);
END left_pad;
```

The LEFT_PAD function will not truncate the string in this call:

```
dbms_output.put_line(left_pad('abcdefg',6,'01234'));
```

This DBMS_OUTPUT displays the original seven-character string—no truncation.

Numeric digits often need formatting. The built-in string functions can be combined to construct functions that provide specific types of padding. The following function uses DECODE, INSTR, and concatenation to return a number as a string, but with two decimal places filled.

```
CREATE OR REPLACE FUNCTION pad_number(N NUMBER) RETURN VARCHAR2
IS
    the_pad    VARCHAR2(3);
    the_number VARCHAR2(30) := TO_CHAR(N);
BEGIN
    SELECT DECODE(INSTR(the_number,'.',1),
         0, '.00',
         LENGTH(the_number)-1, '0',
         LENGTH(the_number)-2, NULL) INTO the_pad FROM dual;
    RETURN '$'||TO_CHAR(N)||the_pad;
END;
```

The PAD_NUMBER function returns the string $99.00 when the input is 99 and returns $99.30 when 99.3 is passed.

Built-in functions can be nested. Padding can be accomplished on both sides of a string by nesting LPAD and RPAD. Take a string and left pad 6 characters with a dash, then right pad that result a full 12 characters with an asterisk.

```
SQL> SELECT RPAD(LPAD('aaa',6,'-'),12,'*') FROM dual;

RPAD(LPAD('A
------------
---aaa******
```

11.13.4 LTRIM, RTRIM

```
LTRIM(str1, [,str2])
RTRIM(str1, [,str2])
```

str1	varchar2	The string to be trimmed
str2	integer	This is one or more characters. TRIM looks for repeated occurrences of this string and removes them. The default is blank.
returns	varchar2	The trimmed string. If the trimming causes all characters to be trimmed, the return is NULL.

The TRIM function removes all repeated occurrences of a string that make up the head or tail of the string. To trim all blanks on either side of a string:

```
LTRIM(RTRIM(string));
```

The LTRIM function removes the first six characters:

```
010101---01---01
```

Once the LTRIM detects an end to the pattern, at the first dash, no other characters are removed.

The SQL statement	Returns
SELECT LTRIM('010101--001--001','01') FROM dual;	--001--001
SQL> SELECT LTRIM('abbba','b') FROM dual;	abbba

SQL> SELECT
    LTRIM(LTRIM(RTRIM('abbba','a'),'a'),'b')
    FROM dual;
	NULL

11.13.5 REPLACE

```
REPLACE(str1, str2 [,str3])
```

str1	varchar2	The string to be modified.
str2	integer	All occurrances of STR2 are replaced with STR3. The string STR2 is assumed to exist at least once

		in STR1.
str3	varchar2	This is the substitution string. It can be larger or smaller than STR2. If STR3 is not provided, then all occurrences of STR2 are replaced with NULL.
returns	varchar2	This could be the original string if no replacement occurred, a modified string, or possibly NULL depending on the substitution.

When writing code that uses REPLACE, also look at the TRANSLATE functions. They each perform string/character substitution. The REPLACE function looks for exact occurrences of STR2. Each exact occurrence is replaced with STR2.

The following replaces the string ABC with XYZ. A letter A, B, or C by itself is unchanged.

```
SQL> SELECT REPLACE ('a b c ab bc abc','abc','xyz') FROM dual;

   REPLACE('ABCABB
   ---------------
   a b c ab bc xyz
```

The TRANSLATE performs a character mapping—a different type of substitution. The following replaces each letter A with X, each B with Y, and each C with Z.

```
SQL> SELECT TRANSLATE('a b c ab bc','abc','xyz') FROM dual;

   TRANSLATE('
   -----------
   x y z xy yz
```

REPLACE can be used to scrub data. To remove all occurrences of a comma followed by a line feed:

```
REPLACE(str, ','||CHR(10))
```

PL/SQL procedures built using the REPLACE function provide powerful data scrubbing techniques. Suppose we want to replace all occurrences of repeated string with a single occurrence. The following procedure takes a base string, and a substring of which repeated occurrences must be replaced with a single occurrence. The result is an IN OUT mode parameter that is the scrubbed string.

```
CREATE OR REPLACE PROCEDURE remove_substring
    (base_string IN VARCHAR2,
     the_unwanted IN VARCHAR2,
     scrubbed_string IN OUT VARCHAR2)
IS
    duplicate VARCHAR2(30) := the_unwanted||the_unwanted;
BEGIN
    scrubbed_string := base_string;
    WHILE
        (REPLACE(scrubbed_string, duplicate, the_unwanted)
         <> scrubbed_string)
    LOOP
        scrubbed_string :=
            REPLACE(scrubbed_string, duplicate, the_unwanted);
    END LOOP;
END remove_substring;
```

This function can be called consecutively each time replacing the BASE_STRING with the result in SCRUBBED_STRING. This PL/SQL block uses REMOVE_SUBSTRING to first replace repeated occurrences of 01 with a single occurrence, then the same for XY.

```
DECLARE
 base_string     VARCHAR2(60) := '01010101-XY-XYXYXYXY';
 clean_this_1    VARCHAR2(2)  := '01';
 clean_this_2    VARCHAR2(2)  := 'XY';
 scrubbed_string VARCHAR2(60);
BEGIN
  remove_substring(base_string,
      clean_this_1, scrubbed_string);
  remove_substring(scrubbed_string,
      clean_this_2, scrubbed_string);
  dbms_output.put_line(scrubbed_string);
END;
```

The final string is: 01-XY-XY

11.13.6 TRANSLATE

```
TRANSLATE(str1, str2, str3)
```

str1	varchar2	The string to be modified.
str2	integer	The first character of STR2 is replaced with the first character of STR3.
str3	varchar2	This is the substitution string. It can be larger or smaller than STR2. If STR3 is not provided, then all occurrences of STR2 are replaced with NULL.
returns	varchar2	This could be the original string if no replacement occurred, a modified string or possibly NULL depending on the substitution.

The TRANSLATE function maps characters from one string onto another string. This is slightly different from REPLACE. To perform the following character mapping:

```
1 -> overwrites each -> A
2 -> overwrites each -> B
3 -> overwrites each -> C

TRANSLATE (string, 'ABC', '123');
```

Using TRANSLATE on a sample string.

```
SQL> SELECT TRANSLATE ('ABC Axy Bxy Cxy CBA','ABC','123')
  2  FROM dual;

TRANSLATE('ABCAXYBX
-------------------
123 1xy 2xy 3xy 321
```

The following overwrites each digit with a zero.

```
TRANSLATE(str,'123456789','000000000');
```

The following translates each digit to a zero, and then removes each zero. This essentially removes all digits.

```
REPLACE(TRANSLATE(str,'123456789','000000000'),'0');
```

The following function returns TRUE if the string contains any digits.

```
CREATE OR REPLACE FUNCTION has_digits
    (str VARCHAR2) RETURN BOOLEAN
IS
    one_to_nine CONSTANT VARCHAR2(9) := '123456789';
    zeroes      CONSTANT VARCHAR2(9) := '000000000';
BEGIN
    -- convert all digits to a 0.
    -- INSTR returns number > 0 of string has a 0.
    RETURN INSTR(TRANSLATE(str,one_to_nine,zeroes),'0') > 0;
END has_digits;
```

11.14 Miscellaneous String Functions

The following list includes common string functions that are used in PL/SQL.

ASCII Returns the decimal representation of a number. The argument is a character. The following SELECT returns 65. The functions ASCII and CHR perform reverse functions.

```
SELECT ASCII('A') FROM dual;
```

CHR Returns the character equivalent of a decimal number. The argument is a decimal number. The result is the ASCII character. The following returns the letter A:

```
SELECT CHR(65) FROM dual;
```

Suppose you want to include the ASCII character (&) in a PL/SQL program. This applies to SQL*Plus. Realizing that (&) is a SQL*Plus special character that denotes a command line argument, you first determine the decimal equivalent.

```
SQL> SELECT ASCII('&') FROM dual;
       38
```

Knowing the decimal equivalent, you can code the following.

```
string_1||CHR(38)||string_2
```

CONCAT The following two expressions are equivalent.

```
var := string_1||string_2
var := concat(string_1, string_2)
```

The first form is the standard form for PL/SQL code. The CONCAT function is used in Pro*C code because the double pipe is compiled as a C operator.

GREATEST Returns the largest value from a set of values. Expressions can be numbers, dates, or character strings.

```
var := GREATEST(var_1, var_2, var_3, etc);
```

LEAST Returns the smallest value from a set.

```
var := LEAST(var_1, var_2, var_3, etc);
```

LENGTH Returns the length of a string. This function works well in conjunction with INSTR, REPLACE, and TRANSLATE.

```
len := LENGTH('PL/SQL'); -- len equals 6
```

INITCAP Returns a string in initial caps.

```
var := INITCAP('NEW YORK');
```

```
var equals 'New York'
```

LOWER Returns a lower case string. The following returns: new york.

```
var := INITCAP('NEW YORK');
```

UPPER Returns an upper case string. The following returns: NEW YORK.

```
var := INITCAP('new york');
```

11.15 Numeric Functions

The following includes common numeric functions you can use in PL/SQL programs.

ABS Returns the absolute value. The argument is any expression that evaluates to a number.

```
var := ABS(var_1 - 100);
```

MOD Returns the remainder following division. The following returns 3.

```
MOD(11,4);
```

ROUND The ROUND function rounds a number or a date. You can specify the decimal place for the rounding. Use a second parameter to indicate the degree of rounding. Positive means round that many places to the right. Negative means round that many places to the left of the decimal point. The following includes some rounding examples.

```
ROUND(199.11);        -- rounds to 200
ROUND(199.11,  1);    -- rounds to 199.1
ROUND(199.125, 2);    -- rounds to 199.13
ROUND(249.11, -2);    -- rounds to 200
ROUND(250.11, -2);    -- rounds to 300
```

SIGN Returns −1, 0, or +1 based on the sign of the expression. The following returns a −1.

```
SIGN(2-5+4*20-100)
```

SQRT Returns the square root of a number. The following returns: 1.4142136.

```
SQRT(2)
```

TRUNC Truncates a number or a date. You can specify the decimal position for truncation. Similar to rounding, you have an optional parameter that is negative or positive. The following are examples.

```
TRUNC(199.99);      -- returns 199
TRUNC(199.99,1);    -- returns 199.9
TRUNC(199.125,2);   -- returns 199.12
TRUNC(249.11,-2);   -- returns 200
TRUNC(299.11,-2);   -- returns 200
```

CEIL Returns the smallest integer greater than or equal to the argument, which is a number. A common numeric calculation is to round up nonwhole numbers. To round up a number with a fraction, you add (0.5) and truncate the number. CEIL does this for you.

```
CEIL(3.0);    -- returns 3
CEIL(3.1);    -- returns 4
CEIL(3.6);    -- returns 4
CEIL(-3.0);   -- returns -3
CEIL(-3.1);   -- returns -3
```

FLOOR Returns the largest integer less than or equal to the argument, which is a number.

```
FLOOR(-3.1);   -- returns -4
FLOOR(-3.0);   -- returns -3
FLOOR(3.1);    -- returns 3
FLOOR(3.6);    -- returns 3
```

POWER Returns a number, raised to this power.

```
POWER(2,10) = 1024
```

LOG(A,X) Common logarithm. This answers the question: A value "A," raised to the power of what is equal to "X"? The LOG functions returns the what.

```
IF A**B=X then LOG(A,X)=B

POWER(2,10) returns 1024
POWER(4,3) returns 64

LOG(2,1024) returns 10
LOG(4,64) returns 3
```

EXP(X) Exponential function of X.

LN(X) Natural logarithm of X.

	EXP(1) = 2.71828183 LN(2.71828183) = 1
COS(X)	Cosine of X.
COSH(X)	Arccosine of X.
SIN(X)	Sine of X.
SINH(X)	Arcsine of X.
TAN(X)	Tangent of X.
TANH(X)	Arctangent of X.

An example of a Pythagorean relation is: For any number X:

POWER(SIN(X),2) + POWER(COS(X),2) = 1

11.16 Random Number Generation

You use the DBMS_RANDOM package to generate random numbers. You set the seed with the SEED function, which is overloaded.

```
PROCEDURE seed(val IN BINARY_INTEGER);
PROCEDURE seed(val IN VARCHAR2);
```

Two methods for setting a seed are:

```
dbms_random.seed(123456);
dbms_random.seed(TO_CHAR(SYSDATE,'dd-mon-yyyy hh24:mi:ss');
```

The function VALUE generates a 38-digit precision number within the range:

```
0.0 <= value < 1.0
```

The function definition for a random number is:

```
FUNCTION value RETURN NUMBER;
```

The following block illustrates the VALUE function.

```
DECLARE
    N NUMBER;
BEGIN
    dbms_random.seed(123456);
    N := dbms_random.value;

    -- N will be a number similar to:
    --
    -- 0.9253168129811330987378779577193159261 8
END;
```

You can generate random numbers within a range. You call the function VALUE and pass the low and high limits. The function definition is:

```
FUNCTION value(low IN NUMBER, high IN NUMBER) RETURN NUMBER;
```

The random number returned is within the range:

```
low <= value < high
```

The following generates 10 random two-digit integer numbers.

```
BEGIN
    dbms_random.seed
      (TO_CHAR(SYSDATE,'dd-mon-yyyy hh24:mi:ss'));
    FOR i in 1..10 LOOP
        dbms_output.put_line(TRUNC(dbms_random.value(10,100)));
    END LOOP;
END;
```

11.17 Date Functions

11.17.1 SYSDATE

```
FUNCTION SYSDATE RETURN DATE;
```

This function returns the current date and time. It can be used in any PL/SQL expressions including initialization of variables. SYSDATE evaluates to a DATE type. If you assign SYSDATE to a string, Oracle will do an implicit conversion.

```
DECLARE
   today DATE := SYSDATE;
BEGIN
   NULL;
END;
```

You can set a default display format for your session with the ALTER SESSION statement.

```
SQL> ALTER SESSION SET
    NLS_DATE_FORMAT='dd-mon-yyyy hh24:mi:ss';
```

The session default display now includes date and time.

```
SQL> select sysdate from dual;

SYSDATE
--------------------
06-may-2004 12:47:07
```

The TRUNC function truncates a date to zero hours, minutes, and seconds. This is the earliest time possible for that day.

```
SQL> select TRUNC(SYSDATE) from dual;

TRUNC(SYSDATE)
--------------------
06-may-2004 00:00:00
```

To see if a date variable precedes the current date, compare it to the truncation of the current day.

```
IF date_variable < TRUNC(SYSDATE) THEN
```

11.17.2 TO_CHAR, TO_DATE

```
FUNCTION TO_CHAR(D DATE [,format_model VARCHAR]) RETURN VARCHAR2;
```

This function is overloaded to convert NUMBER, INTEGER, and other types to character strings. This shows how to use the function to convert a date to a string. You can supply an optional format string with the function. A few format strings are shown here.

Sample Format Model	Output from TO_CHAR
TO_CHAR(SYSDATE,'Day')	Wednesday
TO_CHAR(SYSDATE,'Mon');	Aug
TO_CHAR(SYSDATE,'YYYY');	2003
TO_CHAR(SYSDATE,'Day Month YYYY');	Wednesday August 2003

TO_CHAR(SYSDATE,'DD-MON-YYYY');	06-AUG-2003
TO_CHAR(SYSDATE,'Day Month DD, YYYY');	Wednesday August 06, 2003

```
FUNCTION TO_DATE(V VARCHAR2 [,format_model VARCHAR]) RETURN DATE;
```

This function converts a string to a DATE type. You can use a format model if the string format is not consistent with the DATE format in the database. The following converts a string to a DATE.

```
DECLARE
   D DATE;
   str VARCHAR2(30) := 'Wednesday August 06 2003';
   fmt VARCHAR2(30) := 'Day Month DD YYYY';
BEGIN
   D := TO_DATE(str, fmt);
END;
```

11.17.3 ADD_MONTHS

```
FUNCTION ADD_MONTHS(in_date DATE, months NUMBER) RETURN DATE;
```

This function adds or subtracts one or more months to a date argument.

If today is the last day of the month, such as October 31, and the following month has fewer days, November, this function returns November 30. The following declares DATE variables and initializes them to dates: advanced by one month and one month in the past.

```
same_day_next_month DATE := ADD_MONTHS(SYSDATE, 1);
same_day_last_month DATE := ADD_MONTHS(SYSDATE, -1);
```

11.17.4 LAST_DAY

```
FUNCTION LAST_DAY(in_date DATE) RETURN DATE;
```

This function returns the last day of the current month relative to IN_DATE. The following returns the last day of the current month.

```
v_date := LAST_DAY(SYSDATE);
```

Select all professors who have been hired any time in the current month. If the current day is July 20, we want all rows where

```
hire_date  >= July 1 at time 00:00:00
```

- Subtract a month from the current day with ADD_MONTHS.
- Get the last day of that month with LAST_DAY.
- Add a day, which yields the first day of the current month.
- TRUNC gives time 00:00:00 on the first of the current month.

The beginning of time for the current month is:

```
SELECT TRUNC(LAST_DAY(ADD_MONTHS(SYSDATE, -1))+1)
FROM dual;
```

All professors hired this month are:

```
SELECT prof_name, hire_date
FROM   professors
WHERE  hire_date >=
       TRUNC(LAST_DAY(ADD_MONTHS(sysdate, -1))+1);
```

Let's compute the last day of the previous month.

```
v_date := LAST_DAY(ADD_MONTHS(SYSDATE, -1));
```

The value for V_DATE will include hours, minutes, and seconds, for example:

```
30-JUN-2003 14:32:00
```

If we add a date and truncate, the result is 1-JULY-2003 00:00:00. We can then select all events in the current month with the qualifier:

```
WHERE some_date_column >= TRUNC(v_date+1)
```

We can filter all events previous to the current month with

```
WHERE some_date_column < TRUNC(v_date+1)
```

The beginning of the current month is always:

```
TRUNC(LAST_DAY(ADD_MONTHS(sysdate, -1))+1)
```

11.17.5 MONTHS_BETWEEN

```
FUNCTION MONTHS_BETWEEN (date1 DATE, date2 DATE) RETURN NUMBER;
```

This function returns the number of months between two dates. A fractional part is included in the returned number. The following returns the number of months between January 1 and July 1.

This block assigns N the value of 6.0.

```
DECLARE
   D1 DATE;
   D2 DATE;
   N  NUMBER(4,2);
BEGIN
   D1:= to_date('1-Jul-2004','DD-MON-YYYY');
   D2:= to_date('1-Jan-2004','DD-MON-YYYY');
   N := MONTHS_BETWEEN(D1, D2);  -- N is 6
END;
```

11.17.6 NEW_TIME

```
FUNCTION NEW_TIME (in_date DATE, time_zone VARCHAR2,
    time_zone_of_result VARCHAR2) RETURN DATE;
```

This function evaluates IN_DATE (relative to a time zone—TIME_ZONE) and returns a new time (relative to TIME_ZONE_OF_RESULT).

For example, take the current time zone, assuming Eastern Standard Time, and convert it to Pacific Standard Time.

```
pst_date := NEW_TIME(SYSDATE, 'EST','PST');
```

Convert 12 noon today, Eastern Standard Time, to GMT.

```
DECLARE
  today         DATE:= sysdate;
  converted_time DATE;
BEGIN
    -- set converted_time to 12 noon today.
    converted_time := TRUNC(today) + 1/2;

    -- convert this to GMT time.
    converted_time := NEW_TIME(converted_time, 'EDT','GMT');

    dbms_output.put_line(converted_time);
END;
```

The following table lists the string abbreviations for time conversions.

Time Zone	Conversion String
Atlantic Standard or Daylight Time	AST, ADT
Bering Standard or Daylight Time	BST, BDT
Central Standard or Daylight Time	CST, CDT
Eastern Standard or Daylight Time	EST, EDT
Greenwich Mean Time	GMT
Alaska-Hawaii Standard Time or Daylight Time	HST, HDT
Mountain Standard or Daylight Time	MST, MDT
Newfoundland Standard Time	NST
Pacific Standard or Daylight Time	PST, PDT
Yukon Standard or Daylight Time	YST, YDT

11.17.7 NEXT_DAY

```
FUNCTION NEXT_DAY(in_date DATE, weekday VARCHAR2) RETURN DATE;
```

This function can be used, for example, to return the DATE associated with next Monday. The parameter IN_DATE can be any date. We can compute the first day of a month with LAST_MONTH and LAST_DAY. Incorporation of this function enables computation of the first Monday of a month. The values for WEEKDAY are

- 'MONDAY'
- 'TUESDAY'
- 'WEDNESDAY'
- 'THURSDAY'
- 'FRIDAY'
- 'SATURDAY'
- 'SUNDAY'

What day is the following Monday?

```
v_date := NEXT_DAY(SYSDATE, 'MONDAY');
```

Get the first Tuesday of last month.

```
DECLARE
    start_of_last_month DATE;
    first_tuesday       DATE;
BEGIN
    start_of_last_month :=
            TRUNC(LAST_DAY(ADD_MONTHS(SYSDATE, -2))+1);

    first_tuesday :=
            NEXT_DAY(start_of_last_month, 'TUESDAY');
END;
```

11.17.8 ROUND, TRUNC

```
FUNCTION TRUNC(in_date DATE) RETURN DATE;
FUNCTION ROUND(in_date DATE) RETURN DATE;
```

Time of day starts with hours, minutes, seconds at:

```
00:00:00
```

If you TRUNCATE date, the result is that date at zero hours, minutes, and seconds. This is effectively the start of the day. The following declares a DATE variable and initializes it to the start of the current day.

```
start_of_day DATE := TRUNC(SYSDATE);
```

The start of tomorrow is:

```
start_of_tomorrow DATE := TRUNC(SYSDATE + 1);
```

If a DATE variable is set to a time frame within the current day, the following is TRUE:

```
start_of_day <= variable < start_of_tomorrow
```

Based on this, if you ROUND a DATE the result is one of the following.

```
start_of_today
start_of_tomorrow
```

ROUND always returns the truncation of the current day or the equivalent of the truncation of that day plus 1. If the time is 12 noon or greater, it rounds to the start of the next day.

11.18 Exceptions

There have always been exceptions in programming. The first exception in any computer program was probably a divide-by-zero and at that time there was no mechanism to trap such an exception. A zero divide exception can be captured in PL/SQL with the following. This block prints the error when it occurs.

```
DECLARE
    n NUMBER;
BEGIN
    n:= 2/0;
EXCEPTION WHEN ZERO_DIVIDE THEN
    dbms_output.put_line('Caught a zero divide');
END;
```

The ZERO_DIVIDE is one of the PL/SQL built-in exceptions. The language has several predefined exceptions. We looked at DUP_VAL_ON_INDEX in Chapter 3. This exception is raised from a duplicate insert against a primary key or unique constraint. You can declare your own exceptions. These are called user-defined exceptions.

Excessive use of user-defined exceptions can lead to choppy code. You always want to consider ways to simplify the PL/SQL code that will use your package. A balanced use of user-defined exceptions can simplify the logic, level of nesting, and readability of application code that interfaces with your package specification. In general, user-defined exceptions should be declared and raised within an application when an "exceptional condition" occurs.

11.18.1 User-Defined Exceptions

Exceptions have the following characteristics.

- Exceptions are declared in your program. You can declare an exception in any declarative region: package specification, package body, or subprogram declarative part. The following is an exception you might declare in the specification of a package that contains subprograms to registered students for a class.

  ```
  class_full EXCEPTION;
  ```

- You raise an exception with the RAISE statement. Consider a student registration package (introduced on p. 267). A procedure for adding students would first check if the class is full. The logic of the application to add a student could contain PL/SQL similar to the following:

  ```
  IF num_std >= 30 THEN RAISE class_full; END IF;
  ```

- The user of your package must write code to handle the exception, similar to the ZERO_DIVIDE exception handler.

Style is a concern. There are always options with resolving the unexpected condition. You can return a status to indicate the reason for the unexpected condition. The choice to use an exception has an impact on the code using your package. If your package frequently raises exceptions, the code using your package will be choppy. You should raise an exception only when the exception condition is extremely rare. You can combine exceptions with status returns. For example, you can return a status if the class is full, but raise an exception if the caller attempts to register a student for a nonexistent class.

From an applications perspective, the following package illustrates the use of an exception. This package adds numbers. The package raises an exception if the sum is zero. In this example, a zero is considered an invalid result and the package throws the exception.

```
CREATE OR REPLACE PACKAGE add_pkg IS

    zero_amount EXCEPTION;

    FUNCTION add(A NUMBER, B NUMBER) RETURN NUMBER;
END add_pkg;
```

```
CREATE OR REPLACE PACKAGE BODY add_pkg IS

    FUNCTION add(A NUMBER, B NUMBER) RETURN NUMBER
    IS
        result NUMBER;
    BEGIN
        result := A+B;
        IF (result = 0) THEN raise zero_amount; END IF;
        RETURN result;
    END add;
END add_pkg;
```

The user of the package has a choice in how to write the code. The code can include an exception handler or not include an exception handler. When an exception is raised, execution of the immediate block ends. "Block" means the immediate BEGIN-END block or subprogram body. If the immediate block has an exception handle, then control passes to that point. If there is no exception handler, the exception propagates back to the next-most-outer block or calling subprogram.

The following table shows two procedures; one handles the exception, the other does not.

Procedure with Exception Handler	Procedure without Exception Handling

PROCEDURE proc_1
IS
    N NUMBER;
BEGIN
    N := add_pkg.add(1,2);
    dbms_output.put_line(N);
EXCEPTION
    WHEN add_pkg.zero_amount THEN
        dbms_output.put_line(0);
END;
	PROCEDURE proc_1
	IS
	    N NUMBER;
	BEGIN
	    N := add_pkg.add(1,2);
	    dbms_output.put_line(N);
	END;

11.18.2 Blocks with Exception Handlers

Modify the preceding procedure PROC_1 so that it loops. PROC_1 below calls ADD_PKG once per loop iteration. Theoretically, any iteration can raise an exception. The behavior of this code is that a single exception terminates processing. The first pass will add a minus one to a one. This will raise the exception. The resolution of the exception is to print using DBMS_OUTPUT.

When ADD_PKG raises an exception, the procedure itself, ADD, has no exception handler. Therefore, the exception propagates to the calling procedure, in this case PROC_1. This procedure has an exception handler; hence, control goes to the handler, which prints a zero.

```
CREATE OR REPLACE PROCEDURE proc_1
IS
    N NUMBER;
BEGIN
    FOR VAL IN -1..5 LOOP
        N := add_pkg.add(1,VAL);
        dbms_output.put_line(N);
    END LOOP;
EXCEPTION
    WHEN add_pkg.zero_amount THEN
        dbms_output.put_line(0);
END;
```

The code in PROC_1 can alternatively capture each possible failed addition by embedding the ADD_PKG call in an exception block. This is shown next. This design allows additions to continue while capturing those that fail. The following exception handler exists within a BEGIN-END block. When the exception occurs, control jumps to the end of the current block. The block is within the loop, hence iteration continues.

```
CREATE OR REPLACE PROCEDURE proc_1
IS
    N NUMBER;
BEGIN
    FOR VAL IN -1..5 LOOP
        BEGIN
            N := add_pkg.add(1,VAL);
            dbms_output.put_line(N);
        EXCEPTION
            WHEN add_pkg.zero_amount THEN NULL;
        END;
    END LOOP;
END;
```

11.18.3 The EXCEPTION Clause

The exception clause can be at the end of a procedure, function, or declare block. Previous examples of the procedure PROC_1 illustrate both cases.

The following exception syntax illustrates code that is willing to handle one of two exceptions. Any PL/SQL code can execute within the exception handler.

```
EXCEPTION
    WHEN exception_name THEN do_something;
    WHEN another_exception_name THEN do_something;
END;
```

An exception handler can contain a WHEN OTHERS clause. This program logic handles any exception that occurs.

```
 EXCEPTION
    WHEN OTHERS THEN do_something;
END;
```

A WHEN OTHERS can be combined with other exceptions. Use WHEN OTHERS to catch exceptions not previously listed in the exception handler. The following example prints a specific message when the ZERO_AMOUNT exception is raised. For any "other" exception, a generic message is displayed.

```
EXCEPTION
    WHEN add_pkg.zero_amount THEN
        dbms_output.put_line('Got zero_amount exception');
    WHEN OTHERS THEN
        dbms_output.put_line('Got some other exception');
END;
```

A program can have multiple exception handlers. The next procedure illustrates this. The procedure, PROC_1, has two exception handlers. One exception handler exists within the BEGIN-END block. This captures the ZERO_AMOUNT exception. The other exception handler is at the procedure level and will capture a VALUE_ERROR.

A VALUE_ERROR is an Oracle predefined exception raised on conversion errors such as:

```
N NUMBER := TO_NUMBER('ABC');
```

This version of PROC_1 illustrates exception handling but is also an example of choppy code.

```
CREATE OR REPLACE PROCEDURE proc_1
IS
    N NUMBER;
BEGIN
    FOR VAL IN -1..5 LOOP
        BEGIN
            N := add_pkg.add(1,VAL);
            dbms_output.put_line(N);
        EXCEPTION
            WHEN add_pkg.zero_amount THEN NULL;
        END;
    END LOOP;
EXCEPTION
    WHEN VALUE_ERROR THEN
        dbms_output.put_line('Got value error');
END;
```

Why does PROC_1 have an exception handler for VALUE_ERROR? The only reason is to illustrate the various placements of exception handlers. There is nothing in this code that could possibly raise such an exception. That is, there are no PL/SQL statements that can cause the type of error we would see if we executed this statement:

```
N NUMBER := TO_NUMBER('ABC');
```

Exception handlers that can never be invoked should be removed from the code. But here is a version of PROC_1 that should have a VALUE_ERROR exception handler.

What if we changed PROC_1 as follows? In the following, PROC_1 receives a VARCHAR2 string that is converted to a NUMBER within the loop. The TO_NUMBER conversion could potentially raise a VALUE_ERROR exception.

```
CREATE OR REPLACE PROCEDURE proc_1(END_POINT IN VARCHAR2)
IS
    N NUMBER;
BEGIN
    FOR VAL IN -1..TO_NUMBER(END_POINT) LOOP
        BEGIN
            N := add_pkg.add(1,VAL);
            dbms_output.put_line(N);
        EXCEPTION
            WHEN add_pkg.zero_amount THEN NULL;
        END;
    END LOOP;
EXCEPTION
    WHEN VALUE_ERROR THEN
        dbms_output.put_line('Got value error');
END;
```

Having shown the various places where exception handlers can be declared, the justification for each of these exception handlers is questionable. A better PROC_1 procedure would define the parameter END_POINT as a NATURAL subtype. This constrains the parameter to zero or greater. If this were done, the VALUE_ERROR exception would not be necessary and could then be removed.

11.18.4 SQLCODE and SQLERRM

SQLCODE and SQLERRM are built-in functions that only have meaning within an exception handler. They evaluate to the current exception error number and error message. These functions are frequently used with DBMS_OUTPUT as a debugging mechanism. They are usually in a WHEN OTHERS part of an exception handler.

A procedure with many DML statements can be difficult to debug. There can be many reasons for failure. A procedure may fail because of a duplicate insert. It may fail with a conversion error when it inserts text into a numeric field.

The following example captures an invalid insert. A WHEN OTHERS exception handler is included that prints SQLCODE and SQLERRM.

This block simulates inserting character text into a column with a NUMBER type.

```
DECLARE
    N varchar2(30) := 'not a number';
BEGIN
    UPDATE professors SET salary=TO_NUMBER(n);
EXCEPTION
   WHEN OTHERS THEN
       dbms_output.put_line('SQLCODE:'||SQLCODE);
       dbms_output.put_line('SQLERRM:'||SQLERRM);
END;
```

The output from this block is:

```
SQLCODE:-1722
SQLERRM:ORA-01722: invalid number
```

An INVALID_NUMBER is an Oracle predefined exception raised if conversion fails during a SQL statement. For example:

```
UPDATE professors SET salary=TO_NUMBER('abc');
```

SQLCODE and SQLERRM are usually used in WHEN OTHERS handlers to catch unexpected exceptions. If you know in advance that you will have to catch an INVALID_NUMBER exception, you will code the following:

```
EXCEPTION
   WHEN INVALID_NUMBER THEN
       dbms_output.put_line('Number conversion in SQL');
END;
```

Using %TYPE is a practice that helps avoid run-time-type conversions. To implement the aforementioned PL/SQL block as a procedure that updates the PROFESSORS table, pass the salary using a %TYPE. This eliminates the possibility of an INVALID_NUMBER exception and the need for an exception handler.

```
CREATE OR REPLACE PROCEDURE
    update_salaries(new_sal professors.salary%TYPE) IS
BEGIN
    UPDATE professors SET salary=new_sal;
END update_salaries;
```

11.18.5 The RAISE Statement

An exception condition occurs with the following conditions:

- Application code declares an exception and the PL/SQL code raises that exception. This is the case with the ADD_PKG package code on p. 366. It raises an exception when the result of the operation is zero.
- Oracle raises an exception. This occurs if you insert a duplicate into a primary key column. Another example is a divide by zero.

This section covers the topic of reraising an exception inside an exception handler.

You can capture an exception, handle it, and then raise it. Why? You may want to write exception-handling code for the primary purpose of capturing the error recording it but want to throw it back to the procedure that called you. The procedure that called you may need to know such an exception occurred. Consider the following code, used earlier. This procedure includes the exception handler plus RAISE statement in the BEGIN-END block.

```
CREATE OR REPLACE PROCEDURE proc_1
IS
    N NUMBER;
BEGIN
    FOR VAL IN -1..5 LOOP
        BEGIN
            N := add_pkg.add(1,VAL);
            dbms_output.put_line(N);
        EXCEPTION
            WHEN add_pkg.zero_amount THEN
                dbms_output.put_line('zero_amount');
                RAISE;
        END;
    END LOOP;
EXCEPTION
    WHEN VALUE_ERROR THEN
        dbms_output.put_line('Got value error');
END;
```

What is the execution path when ADD_PKG raises the ZERO_AMOUNT exception?

- First, the exception is originally raised in the ADD procedure of ADD_PKG. This procedure has no exception handler, so control jumps to an exception handler in the block of PROC_1.
- The code in the exception handler is executed. This includes DBMS_OUTPUT. Then there is a RAISE statement. This exits the current block as an exception. The exception propagates to the nearest outer scope block or procedure.
- There is an exception handler for the procedure, but it is specific for a VALUE_ERROR. The RAISE statement within the local BEGIN-BLOCK goes directly to the procedure that called PROC_1.

What if PROC_1 included a WHEN OTHERS exception at the procedure level? In this case, the RAISE statement would be caught in the WHEN OTHER exception part of the procedure exception handler.

Within the body of a PL/SQL block or procedure a RAISE statement must be followed by an exception name. This syntax is:

```
RAISE exception_name;
```

The RAISE statement inside an exception handler may be just the RAISE keyword, that is:

```
RAISE;
```

When RAISE does not identify an exception name, the same exception is raised. However, an exception handler may execute a RAISE statement naming an exception. It can name the same exception of another exception.

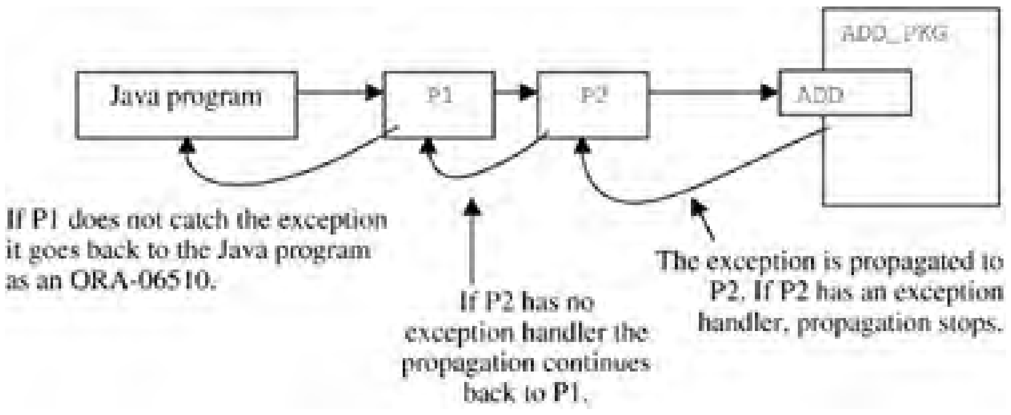
11.18.6 Unhandled Exceptions and Exception Propagation

When an application raises an exception that is not caught, it propagates back to the application as an Oracle error. The Oracle error is:

```
ORA-06510: PL/SQL: unhandled user-defined exception
```

This is a serious problem because it does not indicate the exception name. Consider the scenario in which a Java program calls procedure P1. P1 calls P2, which calls the procedure ADD in the package ADD_PKG. Assume this results in a run-time ZERO_AMOUNT exception. The behavior of the exception propagation is shown in Figure 11-11.

Figure 11-11. Exception Propagation.


ADD_PKG
Java program
P1
P2
ADD
If P1 does not catch the exception it goes back to the Java program as an ORA-06510.
If P2 has no exception handler the propagation continues back to P1.
The exception is propagated to P2. If P2 has an exception handler, propagation stops.

What happens if P2 has any of the following exception handlers:

Exception Handler in P2	Result

EXCEPTION
  WHEN add_pkg.zero_amount THEN
    NULL;
END;
	P1 has no idea an error occurred. From P1's perspective, everything is normal.

EXCEPTION
  WHEN OTHERS THEN
    NULL;
END;
	Same as above.

EXCEPTION
  WHEN add_pkg.zero_amount THEN
    RAISE;
END;
	The exception is propagated back to P1. It may handle it. It may not.

EXCEPTION
  WHEN add_pkg.zero_amount THEN
    RAISE add_pkg.zero_amount;
END;
	Same as above.

EXCEPTION
  WHEN OTHERS THEN
    RAISE;
END;
	Same as above.

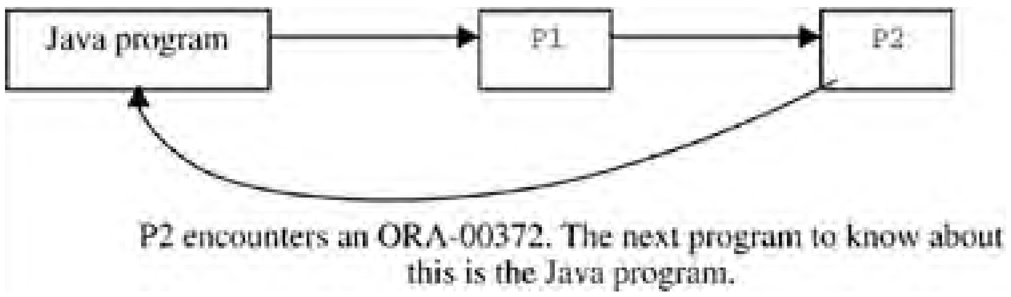
11.18.7 RAISE_APPLICATION_ERROR

RAISE_APPLICATION_ERROR is a procedure in the Oracle built-in package DBMS_STANDARD. Calling this procedure in your code does not raise an exception. This procedure is included under the heading of exceptions because, similar to exceptions, it provides a means to signal an error condition. Exceptions propagate up the PL/SQL call chain. Errors from RAISE_APPLICATION_ERROR do not propagate. The following paragraphs cover the differences between RAISE_APPLICATION_ERROR and exceptions.

A Java application includes error-handling code. The method getErrorCode() is a means by which the Java application realizes that an INSERT or other DML operation failed. There are many reasons why an Oracle error should occur. Some are common, such as a constraint violation, and others occur less frequently, such as a privilege violation. Some are very infrequent, such as writing to a tablespace that is in READ ONLY mode.

Consider the scenario in which a Java program calls a PL/SQL procedure P1. Procedure P1 calls P2. The run-time execution of P2 encounters an error. Procedure P2 tries to write to a READ ONLY tablespace. Control goes to the Java program where it reads the error number as -00372. The error message is: "file 8 cannot be modified at this time." Figure 11-12 shows that behavior of execution.

Figure 11-12. Oracle Errors.


Java program
P1
P2
P2 encounters an ORA-00372. The next program to know about this is the Java program.

PL/SQL exceptions propagate within PL/SQL, but do not propagate outside the language. A PL/SQL exception transforms into the Oracle error:

```
ORA-06510: PL/SQL: unhandled user-defined exception
```

Suppose you want to communicate an error condition from the PL/SQL world to the Java world. You can't raise an exception, but you can "raise an application error" that assumes the behavior of an Oracle error. The interface for this is:

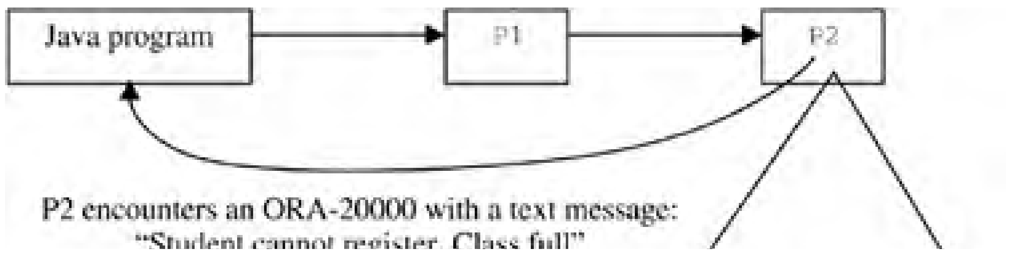
```
RAISE_APPLICATION_ERROR(ARG1, ARG2);
```

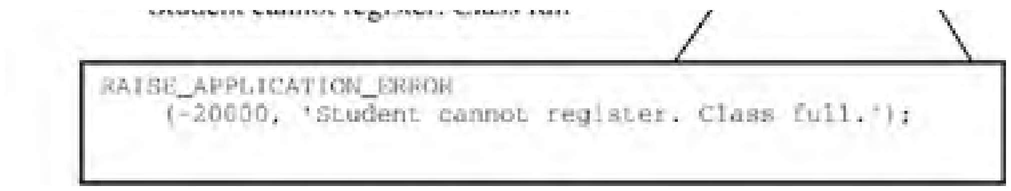
ARG1	Any number within the range (-20999, -20000)
ARG2	Test message indicating your error

As said, the RAISE_APPLICATION_ERROR assumes the behavior of an Oracle error. This means that a Java application can capture your error code number and text message as an application error.

It also means that the execution path immediately leaves the PL/SQL environment and control transfers directly to the calling application, that being the Java, C++, or other language program. In Figure 11-13, procedure P2 raises an error that can be caught by the application. The error number is -20000 and the error text is "Student cannot register. Class is full."

Figure 11-13. Raise Application Error.


Java program
P1
P2
P2 encounters an ORA-20000 with a text message:
"Student cannot register. Class full"



The code in P2 would look like the following:

```
SELECT COUNT (*) INTO number_of_students
  FROM students_courses
 WHERE course_name = v_course_name;

IF number_of_students >= 30 THEN
    RAISE_APPLICATION_ERROR
        (-20000, 'Student cannot register. Class full.');
END IF;
```

11.18.8 EXCEPTION_INIT

Suppose you want to capture an Oracle error, locally. A duplicate insert on a primary key constraint violation raises an exception. That exception can be captured. The error from a check constraint violation does not raise an exception. A check constraint error is an Oracle error. The behavior of an Oracle error is different, as discussed in Section 11.18.7, "RAISE_APPLICATION_ERROR."

In PL/SQL, you can stipulate that a particular error be raised as an exception. To do this you first declare an exception. Then map the exception to the error number. You must know the error number. The PL/SQL statement used is a PRAGMA.

The STUDENTS table has a CHECK constraint on the STATUS column. The column value must be "Degree" or "Certificate." The following procedure can capture a check constraint violation. It declares an exception CHECK_CONSTRAINT_VIOLATION. That is mapped to the error number for the check constraint error. Additionally, this procedure includes exception-handling code for a duplicate insert.

```
CREATE OR REPLACE PROCEDURE
    insert_student(v_stucent_id VARCHAR2,
        v_student_name VARCHAR2,
        v_college_major VARCHAR2,
        v_status VARCHAR2,
        v_state VARCHAR2,
        v_license_no VARCHAR2)
IS
    check_constraint_violation exception;
    pragma exception_init(check_constraint_violation, -2290);
BEGIN
    INSERT INTO students VALUES
        (v_stucent_id, v_student_name,
         v_college_major, v_status,
         v_state, v_license_no);

    dbms_output.put_line('insert complete');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        dbms_output.put_line('PK or unique const violation');
    WHEN check_constraint_violation THEN
        dbms_output.put_line('check constraint violation');
END;
```

Exceptions do not propagate outside of PL/SQL. When unchecked, they become Oracle error:

```
ORA-06510: PL/SQL: unhandled user-defined exception
```

This procedure maps an Oracle check constraint error to an exception. What happens if this procedure raises the exception, CHECK_CONSTRAINT_VIOLATION? That is, what happens if this exception is raised inside of PL/SQL and is uncaught within PL/SQL? The application that called this procedure gets the error code of a check constraint violation, minus 2290. It does not get 06510. It is true that uncaught user-defined exceptions result in Oracle 06510 errors. But Oracle errors mapped to exceptions through PRAGMA EXCEPTION_INIT, when they propagate and are uncaught, result in the original Oracle error.

11.19 Database Access with SQL

11.19.1 Cursor FOR LOOPS

The cursor FOR LOOP requires use of SQL and a PL/SQL FOR LOOP. This is a simple approach to querying the database. You can construct a cursor for loop using any valid SQL statement.

The syntax is:

```
FOR your_name IN (SELECT rest of SQL statement LOOP
    your_name.column_name is defined here
END LOOP;
```

The following block selects all student names and the sum of their parking tickets. The LOOP temporary variable, REC, only exists for the duration of the LOOP. Within the loop we can access columns in the query through the record, REC.

```
BEGIN
    FOR rec IN
        (SELECT student_name, sum(amount) parking_ticket_total
           FROM students a,
                student_vehicles b,
                parking_tickets c
          WHERE a.student_id = b.student_id
            AND b.state=c.state and b.tag_no=c.tag_no
       GROUP BY student_name)
    LOOP
        dbms_output.put_line
            (rec.student_name||' '||rec.parking_ticket_total);
    END LOOP;
END;
```

The aggregate function, SUM, requires a column alias to resolve an attribute name for the record. Literals and expressions must have a column alias. That alias is then used to select the component value from the record.

If the SQL query has a zero result set (i.e., no rows are returned), the loop exits gracefully. There is no exception condition raised.

The cursor can be declared in the declarative part. The loop references the cursor variable.

```
DECLARE
    CURSOR C1 IS
         SELECT student_name, sum(amount) parking_ticket_total
           FROM students a,
                student_vehicles b,
                parking_tickets c
          WHERE a.student_id = b.student_id
            AND b.state=c.state and b.tag_no=c.tag_no
       GROUP BY student_name;
BEGIN
    FOR rec IN C1 LOOP
        dbms_output.put_line
            (rec.student_name||' '||rec.parking_ticket_total);
    END LOOP;
END;
```

11.19.2 Select When Expecting a Single Row

You can expect a single row when the query includes a primary key. The possibility does exist that no row is returned—this would occur if no row exists with that primary key value.

If no rows are returned the NO_DATA_FOUND exception is raised. If there should be multiple rows, the exception TOO_MANY_ROWS is raised.

The only way this would occur would be if the primary key constraint was disabled and duplicate data was loaded.

The following is a stored procedure that selects a single row. If more than one row is returned the exception is raised and caught. If no rows are returned the exception is raised and caught.

```
CREATE OR REPLACE PROCEDURE
    get_student_major
        (v_student_id  IN  students.student_id%TYPE,
         v_name        OUT students.student_name%TYPE) IS
BEGIN
    SELECT student_name
      INTO v_name
      FROM students
     WHERE student_id = v_student_id;
exception
    WHEN TOO_MANY_ROWS THEN dbms_output.put_line('TMR error');
    WHEN NO_DATA_FOUND THEN dbms_output.put_line('NDF error');
END;
```

11.19.3 Inserts and Updates

Inserts and update statements in PL/SQL are very similar to executing SQL in an interactive environment like SQL*Plus. The function SQL%ROWCOUNT is useful after update statements.

SQL%ROWCOUNT	This function evaluates to the number of rows affected by the last INSERT and UPDATE. It equals zero if no rows were changed.

The following function updates professor salaries and returns the number of rows updated. The type of the return argument is NATURAL because the number of updated rows will always be zero or greater.

```
CREATE OR REPLACE FUNCTION
    update_salaries(new_sal IN professors.salary%TYPE)
    RETURN NATURAL IS
BEGIN
    UPDATE professors2 SET salary = new_sal;
    RETURN SQL%ROWCOUNT;
END update_salaries;
```

Use %ROWTYPE to pass multiple components to a subprogram. This procedure inserts a student and accepts a %ROWTYPE as a single parameter. The STUDENT_ID is not included in the original record. That is generated as a sequence number. This procedure inserts the student and returns the STUDENT_ID as part of the record.

A letter "A" is a prefix for the sequence number. The sample data from Chapter 4 includes student ID numbers that begin with a letter. This is in keeping with that convention.

```
CREATE OR REPLACE PROCEDURE
    add_student(rec IN OUT students%ROWTYPE) IS
BEGIN
    SELECT 'A'||students_pk_seq.nextval
      INTO rec.student_id
      FROM dual;

    INSERT INTO students (student_id, student_name,
        college_major, status, state, license_no)
    VALUES (rec.student_id, rec.student_name,
        rec.college_major, rec.status,
        rec.state, rec.license_no);
END add_student;
```

This procedure does an insert. Why does it need to return the record? Why can't REC be IN rather than IN OUT?

It depends on the agreement between the programmer of ADD_STUDENT and those who call this procedure—that may be the same programmer. The calling procedure may need the STUDENT_ID to insert a record in the child table. One option is to pass STUDENT_ID back to the caller using the same record. The IN OUT mode allows the caller to get the STUDENT_ID that was added to the record by this procedure.

An alternative is for the procedure to insert the student and return the STUDENT_ID from a function. This would be a slightly different calling interface. In either case the caller has the new STUDENT_ID for additional use. The following is an interface for inserting a student and a student vehicle. Refer to the DDL in Chapter 4 that shows the data model. In this model STUDENT_VEHICLES is a child to STUDENTS.

A specification for adding student information is the following. This is a package specification that adds a student and adds a student vehicle.

```
CREATE OR REPLACE PACKAGE students_pkg IS
    FUNCTION add_student(rec IN students%ROWTYPE)
    RETURN students.student_id%TYPE;

    PROCEDURE add_vehicle(rec IN student_vehicles%ROWTYPE);
END;
```

The user of this package must first call ADD_STUDENT and then call ADD_VEHICLE. Prior to calling ADD_VEHICLE, the STUDENT_ID must be added to the vehicle record—below this is the assignment prior to calling ADD_VEHICLE. This enforces referential integrity—the STUDENT_ID column is a foreign key to the STUDENTS table.

The user of the preceding package would have PL/SQL logic similar to the following.

```
DECLARE
    student students%ROWTYPE;
    vehicle student_vehicles%ROWTYPE;
BEGIN
    student.student_name  := 'Jack';
    student.college_major := 'HI';
    student.status        := 'Degree';
    student.state         := 'CA';
    student.license_no    := 'MV-232-14';

    student.student_id := students_pkg.add_student(student);

    vehicle.state           := 'CA';
    vehicle.tag_no          := 'CA-1234';
    vehicle.vehicle_desc    := 'Mustang';
    vehicle.parking_sticker := 'A-101';

    vehicle.student_id := student.student_id;

    students_pkg.add_vehicle(vehicle);
END;
```

The body for the STUDENTS_PKG is shown next.

```
CREATE OR REPLACE PACKAGE BODY students_pkg IS

    FUNCTION add_student(rec IN students%ROWTYPE)
        RETURN students.student_id%TYPE
    IS
        ID students.student_id%TYPE;
    BEGIN
        SELECT 'A'||students_pk_seq.nextval INTO ID FROM dual;

        INSERT INTO students (student_id, student_name,
            college_major, status, state, license_no)
        VALUES (ID, rec.student_name, rec.college_major,
            rec.status, rec.state, rec.license_no);

        RETURN ID;
    END add_student;

    PROCEDURE add_vehicle(rec IN student_vehicles%ROWTYPE) IS
    BEGIN
        INSERT INTO student_vehicles (state, tag_no,
            vehicle_desc, student_id, parking_sticker)
        VALUES (rec.state, rec.tag_no,
            rec.vehicle_desc, rec.student_id, rec.parking_sticker);
    END add_vehicle;
END students_pkg;
```

11.19.4 Explicit Cursors

Explicit cursors follow a DO-WHILE-DO loop model; that is:

```
OPEN cursor
LOOP
    FETCH a record
    EXIT WHEN no row returned.
    Process this fetched row.
END LOOP;
CLOSE cursor
```

The cursor-for loop, discussed previously, is a simple approach to querying the database. The cursor-for loop generally has better performance than explicit cursors. However, an explicit cursor may be more appropriate for a particular algorithm. The following discussion shows the main features of an explicit cursor. Then an example is shown.

An explicit cursor requires a cursor definition. This is a SQL statement and a cursor variable. A cursor record structure is declared. The datatype of the cursor record is derived from the cursor definition.

The following PL/SQL block declares a cursor that joins tables STATE_LOOKUP and STUDENTS. To make this interesting, the following features are incorporated into the PL/SQL block:

- REPLACE and NVL functions are used. These functions replace a dash with a space in the student license number. Also, the string "None" is replaced with a NULL should there be no license. A string "N/A" replaces NULL if there is no state.
- The SQL statement in the cursor uses an outer join. This is necessary to include students who have a NULL state column value. Not all students have a license; hence their STATE value is NULL. The STUDENTS is joined with STATE_LOOKUP on the STATE column. The plus operator (+) is appended to the STUDENTS table in the FROM clause.
- The rows are fetched and copied into a global temporary table. After the script, the table is queried for output. The script could use DBMS_OUTPUT, but there is a buffer limitation to DBMS_OUTPUT. The temporary table is more appropriate for spooling large amounts of data.

```
DROP TABLE TEMP;
CREATE GLOBAL TEMPORARY TABLE temp
(name VARCHAR2(10), state VARCHAR2(15), license VARCHAR2(20));

DECLARE
    CURSOR student_cursor IS
    SELECT  a.student_name,
            NVL(b.state_desc, 'N/A') state_desc,
            NVL(REPLACE(a.license_no,'-',' '), 'None') Lic
      FROM  students a, state_lookup b
     WHERE  a.state = b.state(+);

    student_cursor_rec student_cursor%ROWTYPE;
BEGIN

    OPEN student_cursor;
    LOOP
        FETCH student_cursor INTO student_cursor_rec;

        EXIT WHEN student_cursor%NOTFOUND;

        INSERT INTO temp VALUES
            (student_cursor_rec.student_name,
             student_cursor_rec.state_desc,
             student_cursor_rec.lic);
    END LOOP;
    CLOSE student_cursor;
END;

SELECT * FROM TEMP;
```

This PL/SQL block exits the loop on the evaluation of: %NOTFOUND. This cursor attribute is TRUE or FALSE and should be checked after each fetch. An explicit cursor has the following attributes:

%NOTFOUND	This returns TRUE or FALSE based on the last fetch.
%FOUND	The negation of %NOTFOUND.
%ROWCOUNT	This attribute returns the number of rows fetched so far. It can be called anytime after the first fetch. This attribute also returns the number of rows affected from UPDATE and DELETE statements.
%ISOPEN	Returns TRUE if a cursor is still open.

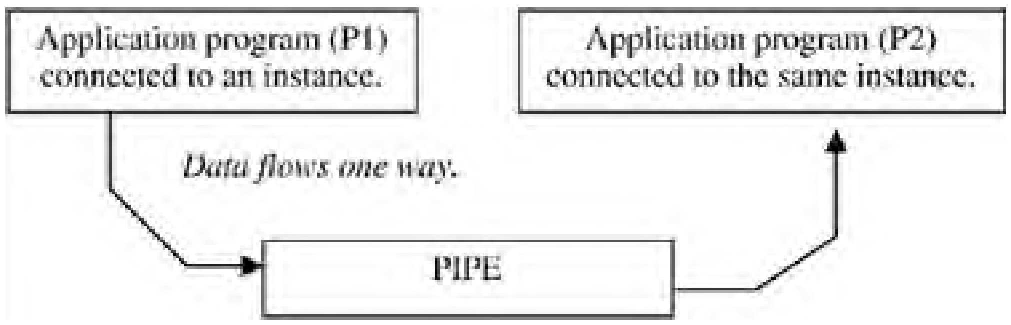
A cursor can be parameter driven. The following block declares a cursor that joins the STUDENTS and STATE_LOOKUP table, but only for students with a particular STATUS. That STATUS is determined when the cursor is opened. This example opens the cursor using a literal string, "Degree."

```
DECLARE
    CURSOR  student_cursor
                (v_student_status students.status%type) IS
    SELECT  a.student_name,
            NVL(b.state_desc, 'N/A') state_desc,
            NVL(REPLACE(a.license_no,'-',' '), 'None') Lic
      FROM  students a, state_lookup b
     WHERE  a.state = b.state(+)
       AND  a.status = v_student_status;

    student_cursor_rec student_cursor%ROWTYPE;
BEGIN

    OPEN student_cursor('Degree');
    LOOP
        FETCH student_cursor INTO student_cursor_rec;

        EXIT WHEN student_cursor%NOTFOUND;

        INSERT INTO temp VALUES
            (student_cursor_rec.student_name,
             student_cursor_rec.state_desc,
             student_cursor_rec.lic);
    END LOOP;
    CLOSE student_cursor;
END;
```

11.20 Sending Pipe Messages (DBMS_PIPE)

Oracle's DBMS_PIPE package is an API to a set of procedures and functions that support communication between two or more processes connected to the same Oracle instance. Oracle pipes are half-diplex. Data flows in one direction. Figure 11-14 illustrates a single pipe for a one-way directional flow of information between two Oracle processes connected to the same instance.

Figure 11-14. Basic Pipe.


Application program (P1) connected to an instance.
Application program (P2) connected to the same instance.
Data flows one way.
PIPE

In Figure 11-14, process P1 sends data to P2. You need a second pipe for P2 to send data back to P1. Consider P1 to be running on a dedicated piece of hardware and connected to an Oracle instance. The other application, P2, could reside on another server. Both applications are connected to the same database instance. The communication is asynchronous. Writers do not wait for readers and readers do not have to wait for writers.

Messages going into the pipe can originate from multiple sources. Remote applications can initiate procedure calls to DBMS_PIPE through ODBC, JDBC, or Net8. Messages can also originate from database triggers. There can be any number of message producers that insert messages into the pipe. Likewise, there can be any number of message consumers that read messages from the pipe. The pipe is a FIFO structure.

Messages sent to a database pipe are not transaction based. The message producer, after sending the message, might do a commit or it might do a rollback; either way, that message remains in the pipe. This mechanism is different from database alerts, implemented with the Oracle DBMS_ALERT package—alerts are transaction based. If a process sends an alert and then rolls back, the alert is voided.

You will be using two subprograms in the package: PACK_MESSAGE and UNPACK_MESSAGE—think of a message part rather than the whole message. A producer can call PACK_MESSAGE 10 times, each time placing a text string into a local buffer. The packing is followed by a single SEND_MESSAGE. The receiver issues one RECEIVE_MESSAGE, which can be followed by 10 UNPACK_MESSAGE calls. Consider the unpacking to be unpacking 10 message parts. The key is that message parts can vary in datatype. One part can be a DATE type; another a VARCHAR2 type.

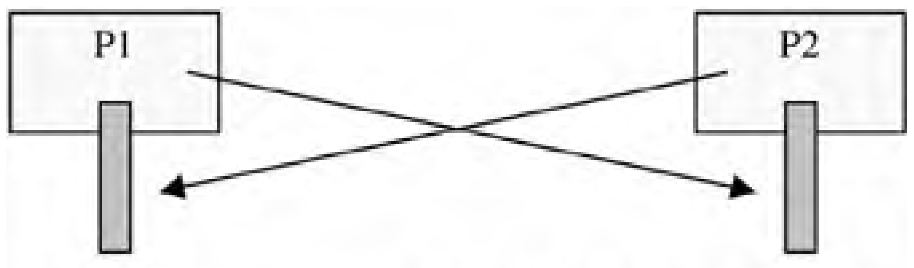
You can combine datatypes to form a single message. The PACK_MESSAGE procedure is overloaded, so is UNPACK_MESSAGE. You can pack a message part of type NUMBER, then a type of VARCHAR2 type, and a DATE. The receiver of a message can "peek" into the pipe message to determine the datatype of the next message part.

11.20.1 Send-Receive Example

The following paragraphs describe and illustrate, with PL/SQL procedures, a design in which processes can communicate in a bidirectional mode. This generalization of communication with pipes can be a basis for solving real-world specific problems with pipes.

A group of processes can communicate with each other by creating a "home" pipe for each process—similar to a mailbox for a house. Each process pulls messages out of their home pipe. Each process knows the name of all other pipes and can send messages to those other home pipes. Figure 11-15 illustrates this communication model.

Figure 11-15. Multiple Processes and Pipes.



P1 sends a message to the home pipe of P2. P2 sends a message to the home pipe of P1.

In a fluid situation, where processes come and go, a software architecture can dynamically generate pipe names with the DBMS_PIPE function, UNIQUE_SESSION_NAME. This function returns a pipe name that is guaranteed to be unique within the instance.

In Figure 11-15, each process has a home pipe for receiving messages. Process P1 can send messages to the pipes of the other processes. The same holds for process P2. When process P1 is not busy, it issues a read on its home pipe. The mechanism for reading a message is to issue a read-wait with a timer expressed in seconds. A process can poll a pipe by setting the timer to zero. Otherwise, a timer in seconds will post a read-wait. A process that receives pipe messages is generally dedicated to servicing pipe messages only. When there is no message in a pipe, the process sits in an idle state waiting for the next pipe message.

The following paragraphs illustrate the code for P1 and P2. In this example, a procedure called P1 builds two message parts and issues a send to a pipe called HOME_OF_P2. A procedure, named P2, reads the message and sends a reply back to P1 using the pipe named HOME_OF_P1. This code demonstrates the communication illustrated in Figure 11-15.

Perform the following steps to demonstrate this communication. First compile and execute the receiver, P2. When you execute P2, the procedure posts a read-wait with a 60-sec. timeout.

The CREATE database script CATPROC.SQL does not grant EXECUTE on DBMS_PIPE to public as it does with DBMS_OUTPUT and many other packages. You need the execute grant to use the package.

1. Connect as SYS and GRANT EXECUTE ON DBMS_PIPE TO SCOTT.
2. Connect as SCOTT.
3. Compile P1 and P2.
4. Connect as SCOTT using SQL*Plus.
5. SET SERVEROUTPUT ON.
6. Execute P2—your session will hang because you have issued a read-wait on your home pipe. The read-wait is for 60 sec. If no message is sent from P1 in that time, there will be a harmless error generated—we'll talk about pipe errors shortly. Before the 60 sec is up, you need to perform the next three steps.
7. Start a new SQL*Plus session.
8. SET SERVEROUTPUT ON.
9. Execute P1—this starts the process by sending a message to the home of P2. After sending the message, the procedure issues a read-wait (for 60 sec) on its home pipe.

The following is the PL/SQL code for procedures P1 and P2.

```
CREATE OR REPLACE PROCEDURE P1
is
   status      integer;
   response    varchar2(2000);
begin
   dbms_pipe.reset_buffer;
   dbms_pipe.pack_message('This is message 1');
   dbms_pipe.pack_message('This is message 2');
   status := dbms_pipe.send_message('HOME_OF_P2');
   status := dbms_pipe.receive_message('HOME_OF_P1', 60);
   dbms_pipe.unpack_message(response);
   dbms_output.put_line('P1 received:'||response);
```

```
end P1;

CREATE OR REPLACE procedure P2
is
    status       integer;
    message_1    varchar2(2000);
    message_2    varchar2(2000);
begin
    status := dbms_pipe.receive_message('HOME_OF_P2', 60);
    dbms_pipe.unpack_message(message_1);
    dbms_pipe.unpack_message(message_2);
    dbms_pipe.pack_message('Got your messages.');
    status := dbms_pipe.send_message('HOME_OF_P1');
    dbms_output.put_line('P2 received:'||message_1);
    dbms_output.put_line('P2 received:'||message_2);
end P2;
```

The session output for the P2, which is executed first, is the following.

```
SQL> set serveroutput on
SQL> execute p2
P2 received:This is message 1
P2 received:This is message 2
```

The session output for the P1 is:

```
SQL> set serveroutput on
SQL> execute p1
P1 received:Got your messages.
```

11.20.2 Interface Description

The following paragraphs describe the DBMS_PIPE package API.

PACK_MESSAGE

```
PROCEDURE pack_message(item in VARCHAR2);
PROCEDURE pack_message(item in NUMBER);
PROCEDURE pack_messge(item in DATE);
```

item	This is a VARCHAR2, NUMBER, or DATE type variable that is placed into a local buffer. You can pack a single buffer with several messages, each of a different type.

The DBMS_PIPE package provides overloaded procedures for packing different datatypes. You can pack RAW and LONG datatypes. The model for sending data to a pipe is to PACK one or more times, then SEND once.

The PACK_MESSAGE procedure copies data to a process-specific Oracle memory buffer. Each Oracle connection has a buffer available for 4096 bytes of data. This buffer can be packed in stages, from different programs within a single Oracle connected session. Packing a message is not the same as sending a message to a pipe. Several messages can be packed and be followed by a single send, which will place all packed messages in the destination buffer. Once a buffer has messages from a PACK_MESSAGE call, there are three events that can occur:

- The process disconnects from Oracle and no messages are sent.
- The process calls DBMS_PIPE.RESET_BUFFER, which clears the buffer of all messages.
- The process calls DBMS_PIPE.SEND_MESSAGE, which sends all messages and clears the buffer.

If you PACK without interruption, you will overflow the 4096 limitation. This overflow condition generates an ORA-06558 error. Similarly, when we UNPACK a buffer we can unpack an empty buffer and get an underflow condition.

When the contents of a buffer are sent to a pipe, that buffer is cleared. For example, you can pack 4096 bytes into your session buffer and send these bytes to a pipe. You can call PACK_MESSAGE again, packing another 4096 bytes. This second buffer can be sent. You have to be careful; you have physical limitations when sending data to a pipe. When you send a full buffer to a pipe and the receiver has not read its home pipe, your next send will have to wait for the receiver to read its home pipe.

SEND_MESSAGE

```
FUNCTION send_message(
    pipename in VARCHAR2,
    timeout IN INTEGER DEFAULT maxwait,
    maxpipesize IN INTEGER DEFAULT 8192) RETURN INTEGER;

    maxwait constant integer := 86400000;  /* 1000 days */

    RETURN INTEGER values:

    0 – Success.
    1 – Timeout because the pipe stays full leaving no room, or
        the sending process cannot place a lock on the pipe.
    3 - Interrupted.
```

pipename	This parameter is the name of the pipe that is to receive all messages present in the local buffer. Pipe names are limited to 128 characters. Do not use pipe names beginning with ORA$ as these are reserved by Oracle.
timeout	This is an optional argument that is the time allowed for moving the buffer contents to the pipe. This is not the time limit placed on a process reading the message. If a process is slow in reading messages from its home pipe (i.e., slow compared to the rate at which data is flowing into the pipe, the pipe may be full; hence a SEND_MESSAGE will wait for room to be made. If room in the pipe is not freed, the timer will expire and the message is not sent.
maxpipesize	This is the maximum size message allowed for the pipe. This must be at least as large as the message. Subsequent calls to SEND_MESSAGE with a larger maxpipesize will increase the maximum size of a message allowed in the pipe. Maxpipesize, when used, becomes a persistent attribute of the pipe. The demo procedures above, P1 and P2, used this parameter. Had those procedures explicitly created their pipes with the CREATE_PIPE procedure, which requires a maxpipesize, then there would not be a need for maxpipesize as an argument to SEND_MESSAGE.

The SEND_MESSAGE procedure copies the contents of the message buffer into a designated pipe. Upon completion, there is no guarantee that the message has been read. A successful return code indicates only that the message was copied. After the call, the local message buffer is empty.

RECEIVE_MESSAGE

```
FUNCTION receive_message(
    pipe_name IN VARCHAR2,
    timeout IN INTEGER DEFAULT maxwait) RETURN INTEGER;

    maxwait constant integer := 86400000;  /* 1000 days */

    RETURN INTEGER values:

    0 - Success
    1 – Timeout
    2 – Record in pipe is too large for the buffer.
    3 - Interrupted.
```

pipename	This parameter is the name of the pipe that is the source of the message to be received. In previous discussions, this was referred to as the home pipe. Pipe names are limited to 128 characters. Do not use pipe names beginning with ORA$ as these are reserved by Oracle.
timeout	An optional argument is the time allowed for moving the pipe contents to the buffer. If the pipe is empty you will wait on a message until the timer expires. You can post a read on just one pipe. This is in contrast to alerts (DBMS_ALERT package) where you can post a "wait on any" alert. If a process wishes to receive data from several pipes, that process must pole each pipe with a RECEIVE_MESSAGE. For this reason, there is no advantage to a design in which a process has several home pipes. You can use a timeout of zero to issue a nonblocking read.

The RECEIVE_MESSAGE function copies messages from the pipe into the local buffer. A successful read is indicated with a zero return code. If no messages are present in the pipe, you still get a zero, indicating a successful read. Once you issue RECEIVE_MESSAGE, you then unpack the message from the local buffer into local variables.

UNPACK_MESSAGE

```
PROCEDURE unpack_message(item OUT VARCHAR22);
PROCEDURE unpack_message(item OUT NUMBER);
PROCEDURE unpack_messge(item OUT DATE);
```

Item	This is a variable that has been declared in your procedure.

The DBMS_PIPE package provides overloaded procedures for unpacking different datatypes. You can pack RAW and LONG datatypes. The model for receiving data is to issue one RECEIVE followed by a series of UNPACK commands.

The UNPACK_MESSAGE procedure copies the next message from your buffer into the variable. This is an OUT mode variable. Following the call to UNPACK, that variable contains the contents of the latest message. How do you know what, if anything, is in the buffer, especially because a RECEIVE_MESSAGE returns a successful return code when the pipe is empty? The function NEXT_ITEM_TYPE is the key. If your code needs to loop over messages in the buffer, you want to incorporate NEXT_ITEM_TYPE in your loop. This is illustrated in the following sample code.

RESET_BUFFER

```
PROCEDURE reset_buffer;
```

The RESET_BUFFER procedure initializes buffer position variables declared within the package body of DBMS_PIPE. When a message is sent with SEND_MESSAGE, these same variables are initialized as a method of clearing, and resetting, the internal buffer. You would call this procedure if a buffer was packed and you wanted to clear it and repack it with different messages, prior to sending the buffer messages to a pipe.

PURGE

```
PROCEDURE purge(pipename IN VARCHAR2);
```

pipename	The name of the pipe you are cleaning out.

The PURGE procedure removes any messages in the pipe. This procedure frees all memory associated with the pipe. Processes that send and receive messages can abort. When this occurs, and the processes are restarted, they may see messages left in the pipe from the previous run. If this scenario is possible, the logic of these processes should be able to detect, through the presence of messages in the pipe, a warm start situation, as opposed to a cold start.

NEXT_ITEM_TYPE

```
FUNCTION next_item_type RETURN INTEGER;

  RETURN INTEGER values:

  0 – No more items in the pipe.
  9 – The next item in the buffer is a VARCHAR2
  2 – The next item is a NUMBER.
  3 – The next item is a DATE.
```

The NEXT_ITEM_TYPE function provides the ability to "peek" into the message buffer. It is useful when messages of various types are being sent. Even if you know what type of message is being sent, this function can be incorporated into a WHILE LOOP so you can break easily when the message buffer is empty.

If you do not use this function to detect an empty buffer, then you risk calling UNPACK_MESSAGE on an empty buffer—this will generate an ORA-06556 or an ORA-06559 error.

UNIQUE_SESSION_NAME

```
FUNCTION unique_session_name RETURN VARCHAR2;
```

The UNIQUE_SESSION_NAME function returns a name like ORA$PIPE$ followed by a sequence of letters and digits. This pipe name is provided by Oracle and is guaranteed to be unique throughout all sessions that connect to that instance. A process must know the name of a pipe prior to calling SEND_MESSAGE; the same holds true for the function, RECEIVE_MESSAGE. When UNIQUE_SESSION_NAME is used to generate pipe names, there has to be some central pipe name registration in the form of a database table where applications register their pipe names; then every process can know the home pipe name of all other processes.

CREATE_PIPE, REMOVE_PIPE

Creation and removal of a pipe can be accomplished with the following two functions.

```
FUNCTION create_pipe(
    pipename IN VARCHAR2,
    maxpipesize IN INTEGER DEFAULT 8192,
    private IN BOOLEAN DEFAULT TRUE) RETURN INTEGER;

FUNCTION remove_pipe (pipename IN VARCHAR2) RETURN INTEGER;
```

Our examples use the functions SEND_MESSAGE and RECEIVE_MESSAGE to implicitly create the pipes. These functions create the pipe named in the call if it does not already exist. You can implicitly create a function and yet still remove it with REMOVE_PIPE.

11.20.3 Exception Handling

The following lists the possible exceptions that can occur. The last item in this list, UNPACK_MESSAGE, is the most common, and we will demonstrate PL/SQL code to handle this condition.

- SEND_MESSAGE: The destination pipe may remain full. When this happens, you will wait for TIMEOUT seconds to pass and then get a return code of 1.
- RECEIVE_MESSAGE: If the message in the pipe is too large for the buffer, you get a return code of 2. This event is not likely because all messages originate from the same size buffer, which is limited to 4096. The pipe limit is twice that.
- PACK_MESSAGE: You can overflow your local buffer, which produces an ORA-06558 error.
- UNPACK_MESSAGE, NEXT_ITEM_TYPE: You can unpack and empty your buffer, which produces an ORA-06556 or an ORA-06559 error.

In PL/SQL code, you can use PRAGMA EXCEPTION_INIT to ensure that, should a particular ORA error occur, you capture that as an exception. This is how we capture an Oracle error.

We are most concerned with an error when we unpack an empty message. If no message has been sent to our pipe, the RECEIVE_MESSAGE returns a status of 0. The following error is generated when we follow that receive with either a call to UNPACK_MESSAGE or NEXT_ITEM_TYPE

```
ORA-06556: the pipe is empty, cannot fulfill
the unpack_message request
```

The following procedure, P3, is more robust because it will not fail. The parsing of message parts is embedded within a loop—this permits us to get all message parts. That code is further embedded within an exception handler. All of this logic is embedded within a loop—but one that loops three times. Procedure P3, shown next, illustrates how a PL/SQL procedure can serve as an endless-loop message handler.

```
CREATE OR REPLACE procedure P3
is
    status          integer;
    message_part      varchar2(2000);
    message          varchar2(2000);
    empty_buffer      exception;
    pragma exception_init(empty_buffer, -6556);
begin
    for i in 1..3 loop
        status := dbms_pipe.receive_message('HOME_OF_P3', 5);

        begin
            while (dbms_pipe.next_item_type = 9) loop
                dbms_pipe.unpack_message(message_part);
                message := message||'-'||message_part;
            end loop;
        exception
            when empty_buffer then null;
        end;
    end loop;
    dbms_output.put_line(message);
end P3;
```

Procedure P4 is demonstrated with the following SQL*Plus session, which sends messages to the pipe of P3. When P3 completes three cycles, the output is the concatenation of whatever messages were read. For the SQL*Plus session we have (start P3 before this session):

```
SQL> set feedback off
SQL> variable status number
SQL> execute dbms_pipe.reset_buffer;
SQL> execute dbms_pipe.pack_message('This is message 1');
SQL> execute dbms_pipe.pack_message('This is message 2');
SQL> execute :status := dbms_pipe.send_message('HOME_OF_P3');
```

The output from P3 is the following.

```
SQL> SET SERVEROUTPUT ON
SQL> execute p3
-This is message 1-This is message 2
SQL>
```

11.21 Signaling Events with Alerts (DBMS_ALERT)

The Oracle DBMS_ALERT package allows a process to post a message to another process and have that message delivered only after a commit. This is different from pipes where a process posts a message and that message will always be sent to the pipe, regardless of commit or rollback.

Pipe messages are sent immediately. Alerts are delivered only after the commit. A process can post many alerts. None can be received until the sending process has done a commit.

Alerts are asynchronous. The sender can send an alert signal whether or not a receiver has a posted read. This is also true with pipes. Receivers can post a read-wait with a timer or they issue a read and return. Senders and receivers communicate with the alert buffer independent of each other. Senders never wait on receivers, or vice versa.

Alerts are for processes connected to the same instance. The following is one scenario.

An application program, APP_A, written in Java and running on server host01.domain.com, is connected to a database on hostdb.domain.com through JDBC. A second application, APP_B, written in C, is running on host02.domain.com—this is connected to the same database on hostdb.domain.com with ODBC. Application APP_A can post alerts using the DBMS_ALERT package. Application APP_B, running on a separate server, can receive those alerts—the intermediary is the database. This scenario is essentially application-to-application communication, even though these applications run on separate machines.

Alerts can also be fired from triggers. That is a practical form of asynchronous communication because you can post numerous alerts, but should the transaction fail, the alerts are never delivered. This is not so with pipes. Messages posted to a pipe from within a database trigger will always be delivered. This behavior helps when deciding whether to use a pipe or an alert. If you are sending text as a general means of communication and that communication is not tied to a transaction, then use a pipe. If it's transaction based, use an alert.

Pipes are also more suited for larger text messages. Alerts are modeled after the UNIX SIGNAL paradigm, which is intended to just signal an event. One process sends a signal to another process, and the mere name of that signal, in this case alert name, indicates the type of event that took place.

The following summarizes the differences between alerts and pipes.

- Use pipes for exchanging textual information because you can pass VARCHAR2, NUMBER, and DATE types, and the buffer for transmission is 4K.
- Use alerts to signal an event. The alert name is the indicator of the particular event that took place.
- Alerts are limited in textual transmission. You can only deliver 1,800 characters of text. The alert name is what identifies the event.
- A message posted to a pipe is always delivered. When you post an alert, you are saying, "allow the receiver to get this only if and when I commit."

Alerts and pipes can work together. Two processes can post messages to pipes but use alerts as an indicator that work needs to be done. For example, a worker-process may spend most of its processing time idle, waiting to receive any one of several alert signals. When that process receives an alert, that particular alert indicates a specific task. Based on that task, the worker-process will read messages from one of several pipes.

Messages sitting in pipes and alerts, waiting to be delivered, are temporary. If the database goes down, the alerts and pipe messages will not be present when the database comes up. The Oracle Advanced Queuing option provides a robust form of queuing messages that are integrated into the database. Queues are implemented with tables, so queues are restored during crash recovery. If you put a message in a queue and the database goes down before the queue message is delivered, the queue message will be present when the database comes back up. The API to queues is extensive, compared to pipes and alerts. The pipes and alerts are straightforward forms of communication, but a message or alert can potentially be lost if the database crashes before a message can be received.

Pipe communication requires no synchronization between a sender and receiver. A PL/SQL procedure can send a message to a pipe at any time. Whether a receiver procedure reads from that pipe has no effect on the sender. There is a mild synchronization required with alerts. A receiver must first register for an alert. A receiver can register for many alerts, but someone must register first. When the alert is sent the process that registered will be able to get the alert. Once an alert is sent, it is too late to register and receive. The register must occur first. If you send an alert and no process has registered for that alert, no process will ever be able to receive it.

"Event" and "Signal" are excellent synonyms for an alert. If we want to know that an event took place, we only need to be told once. "The light is on"—that is an event, or an alert. If 10 people say the light is on, the message is the same as if said once. A process can send the same alert twice, or even three times. When a receiving process issues a read, Oracle will deliver a single alert because the receiver only needs to be told once that that event has occurred. Messages, up to 1,800 characters, can accompany an alert. When multiple alerts with the same name are issued and there is one delivery, the message with the last sent alert is the message delivered.

11.21.1 Interface Description

The following paragraphs describe the DBMS_ALERT package API.

SIGNAL

```
PROCEDURE signal(
    name IN VARCHAR2,
    message IN VARCHAR2);
```

name	The name of the alert. Alert names are best if they indicate the particular event they signal, such as LOW_INVENTORY. Alert names are limited to 30 characters.
message	This is supplemental text to the alert. The limit is 1,800 characters. You can only send a VARCHAR2 as additional text. You cannot send a NUMBER or DATE type variable. This procedure is not overloaded with other types.

This procedure sends an alert. Unless some other process has previously registered for an alert, this call is meaningless. Some processes must first register for an alert—then a sender can send that alert.

The alert name is the key component—this indicates the event that needs to be transmitted. You may find that you rarely use the message parameter, or use it for supplemental information only. Suppose you have a manufacturing system. In this system, inventory is moved from a stock warehouse to an assembly floor—this is a fully automated system. The movement of materials involves a DELETE trigger on a database inventory table. When items are deleted, the trigger evaluates how much inventory is left. If inven-tory is too low, resulting from the delete, the trigger sends a LOW_INVENTORY alert to another ORDER INVENTORY application. That alert name should be sufficient information as to what needs to be done. The additional 1,800 characters can be supplemental information that specifies the specifics of exactly what materials are short.

REGISTER

```
PROCEDURE register(name in VARCHAR2);
```

name	The name of the alert. Alert names need to be synchronized. Because all processed communication with alerts must be connected to the same database instance, the official list of alert names can be stored in a database table.

The general behavior of a receiver is to register for one or more alerts and then wait on any of those alerts. Let's continue with the manufacturing example. There are a variety of critical events in a manufacturing environment. You can have an application that remains in an idle state, waiting for a problem to solve (e.g., problems such as low inventory and out-of-stock items). This application would begin by registering for both events and then posting a read-wait on either of these events. This PL/SQL code would begin with the following.

```
dbms_alert.register('LOW_INVENTORY');

dbms_alert.register('OUT_OF_STOCK_ITEM');

dbms_alert.waitany(name, message, status);
```

This application would "sit" on the WAITANY call until an alert was received.

WAITANY

```
PROCEDURE waitany(
    name OUT VARCHAR2,
    message OUT VARCHAR2,
    status OUT INTEGER,
    timeout IN INTEGER DEFAULT maxwait);

    maxwait constant integer := 86400000;  /* 1000 days */
```

name	This is an OUT mode parameter. The first order of business for a procedure that uses WAITANY is to determine the value of this variable. This variable will contain one of the alert names for which the process is registered.

message	This is an OUT mode parameter used if the alert sender wishes to deliver supplemental text. This is limited to 1,800 characters.
status	0 is success. Status = 1 when the timer expired. So if timeout is 10 and you return with a status of 1, then no alerts were there to receive and the 10-sec timer expired.
timeout	This is a read-wait timer in seconds. This has a default of 1,000 days.

This procedure is very useful. It means that a single process can register for multiple events and then wait, with a single call, on the occurrence of any event. When the event occurs (i.e., the alert is received), the process performs some work, then goes back and issues another wait.

A process cannot restrict a WAITANY to a subset of the alerts for which that process is registered. If a process registers for a LOW_INVENTORY, OUT_OF_STOCK_ITEM, and other alerts, then a WAITANY will always, and can only, wait on that full set of alerts.

Alerts that are sent within a transaction that is uncommitted do not block subsequent alerts that are sent in committed transactions. Assume we have a process that waits on LOW_INVENTORY and OUT_OF_STOCK_ITEM. Another process sends the alert, LOW_INVENTORY, but has not done a commit. Seconds later another process sends the alert OUT_OF_STOCK_ITEM with an immediate commit. The process with the WAITANY call will receive the OUT_OF_STOCK_ITEM immediately.

WAITONE

```
PROCEDURE waitone(
   name IN VARCHAR2,
   message OUT VARCHAR2,
   status OUT INTEGER,
   timeout IN INTEGER DEFAULT maxwait);
```

name	This is the name of the alert for which you wish to wait. The other parameters are identical to the WAITANY procedure described earlier.

If an application is registered for several alerts, it makes sense to post a WAITANY. The WAITONE procedure is for posting a read for a specific alert. If you register for just one alert, this procedure can be used to wait on that specific. The choice between WAITONE and WAITANY is a design issue and depends on how many alerts upon which you want to post concurrent read-waits.

REMOVE

```
PROCEDURE remove(name IN VARCHAR2);
```

Oracle documentation recommends that a procedure remove an alert when it no longer needs that alert. An application program may be designed to always have a posted read on one or more alerts. In this case, that application is never "finished" with any of its registered alerts. This is quite acceptable. A procedure can remove all registered alerts with the REMOVEALL procedure.

REMOVEALL

```
PROCEDURE removeall;
```

An application can remove all alerts for which it is registered with this one call.

SET_DEFAULTS

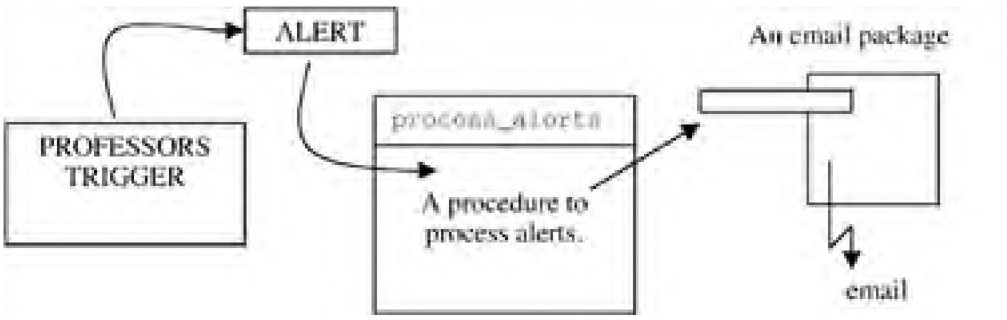
```
PROCEDURE set_defaults(sensitivity IN NUMBER);
```

This procedure sets a polling "sleep time" value that is used internally with calls to WAITANY. It is not necessary to use this procedure unless you want to override the default sleep period, which is five sec.

11.22 Email Notification with Triggers and Alerts

Database triggers can interface with database pipes and alerts. Figure 11-16 illustrates a model in which a database trigger posts a notification that a professor's salary has changed. This is a trigger that fires only from an update of the row. The trigger posts the alert. When the transaction commits, the signal is received by a second Oracle database connection. This connection is represented in Figure 11-16 as PROCESS_ALERTS, which has one purpose, to deliver email.

Figure 11-16. Trigger Email Notification.



Because PROCESS_ALERTS runs asynchronously, it has no impact on other database activity. Update transactions to the PROFESSORS table do not wait for an email to be sent.

We start with developing an interface that will service email requests. This interface will be used by PROCESS_ALERTS. It will accept standard email parameters: sender, receiver, subject, and text. This interface is a package and has the following specification:

```
CREATE OR REPLACE PACKAGE email_pkg IS
    PROCEDURE send
        (p_sender IN VARCHAR2, p_recipient IN VARCHAR2,
         p_message IN VARCHAR2, p_subject IN VARCHAR2);
END email_pkg;
```

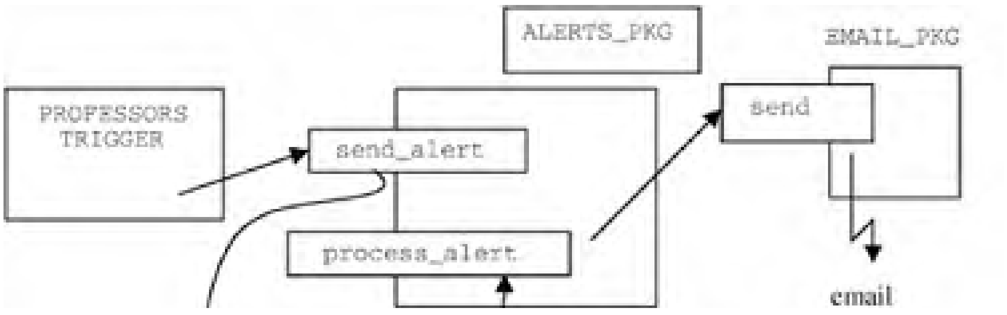
The next step is to develop an engine that will dedicate itself to servicing signals. This can be a stand-alone procedure. Figure 11-16 shows PROCESS_ALERTS as a stand-alone procedure. At this point we need to consider using a single procedure, and a package could be a better choice.

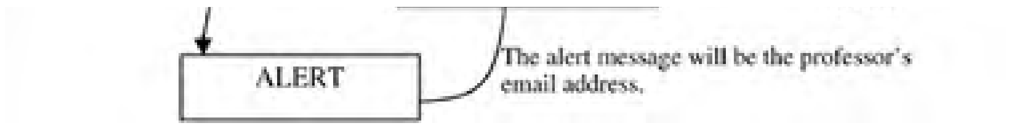
We need a procedure to receive alerts and a procedure to send alerts. The sending occurs in the trigger. It seems reasonable to define a package specification to support the send and receive functions. That package specification is shown next.

```
CREATE OR REPLACE PACKAGE alerts_pkg IS
    PROCEDURE process_alerts;
    PROCEDURE send_alert(message IN VARCHAR2);
END alerts_pkg;
```

Figure 11-16 is redrawn to show the modified architecture. In Figure 11-17, the trigger calls the SEND_ALERT procedure to post the alert. The code used to receive the alert is in the same package.

Figure 11-17. Revised Trigger Email Notification.





The assumption is that the trigger will use the professor's name to construct an email address and pass that address to the procedure in the ALERTS_PKG package. Ideally, the PROFESSORS table would have a column that contains email addresses.

The database trigger is set up to send an email only when there is a difference in the old and new salary.

```
CREATE OR REPLACE TRIGGER professors_aur
AFTER UPDATE ON professors
FOR EACH ROW
WHEN (OLD.SALARY <> NEW.SALARY)
BEGIN
    alerts_pkg.send_alert(:new.prof_name||'@domain.com');
END;
```

For this model, all interfaces have been shown. We can start looking at the body for the individual packages. The email body is shown here. This body includes global declarations for the SMTP server IP address and port number. This is the mechanism by which the package ALERTS_PKG will deliver email for each alert received. The body of ALERTS_PKG will include a call to the email SEND procedure.

```
CREATE OR REPLACE PACKAGE BODY email_pkg IS
    g_smtp_server      CONSTANT VARCHAR2(20) := '00.00.00.00';
    g_smtp_server_port CONSTANT PLS_INTEGER := 25;

    PROCEDURE send
        (p_sender IN VARCHAR2, p_recipient IN VARCHAR2,
         p_message IN VARCHAR2, p_subject IN VARCHAR2)
    IS
        mail_conn   utl_smtp.connection;
    BEGIN
        mail_conn := utl_smtp.open_connection
            (g_smtp_server, g_smtp_server_port);

        utl_smtp.helo (mail_conn, g_smtp_server);
        utl_smtp.mail (mail_conn, p_sender);
        utl_smtp.rcpt (mail_conn, p_recipient);
        utl_smtp.open_data(mail_conn);

        utl_smtp.write_data
            (mail_conn,'From: "'||p_sender
             ||'" <'||p_sender||'>'||utl_tcp.CRLF);
        utl_smtp.write_data
            (mail_conn,'To: "'||p_recipient
             ||'" <'||p_recipient||'>'||utl_tcp.CRLF);
        utl_smtp.write_data
            (mail_conn, 'Subject: '
             ||p_subject||utl_tcp.CRLF);
        utl_smtp.write_data
            (mail_conn, utl_tcp.CRLF||p_message);
        utl_smtp.close_data(mail_conn);
        utl_smtp.quit (mail_conn);
    END send;
END email_pkg;
```

The package body for the sending and receiving of alerts is shown next. The subprogram to receive alerts is coded to wait for three alerts; each wait includes a 10-sec timer. The loop also terminates when it receives an alert message of "END." The alert device name is "email_notification."

For an anachronous application, a separate process that runs in the background will invoke PROCESS_ALERTS. Locally, PROCESS_ALERTS can be run from SQL*Plus. As coded here, it will deliver the first three emails that result from updates to the PROFESSORS table.

```
CREATE OR REPLACE PACKAGE BODY alerts_pkg IS
    PROCEDURE process_alerts
    IS
        professor_email VARCHAR2(100);
        status          INTEGER;
    BEGIN
        dbms_alert.register('email_notification');
        FOR I IN 1..3 LOOP

            dbms_alert.waitone


                (name    => 'email_notification',
                 message => professor_email,
                 status  => status,
                 timeout => 10);

            IF status = 0 THEN
                EXIT WHEN professor_email = 'END';

                email_pkg.send
                  (p_sender=>'admin@school.com',
                   p_recipient=>professor_email,
                   p_subject=>'Salary',
                   p_message=>'Salary has changed');
            END IF;
        END LOOP;
    END process_alerts;

    PROCEDURE send_alert(message IN VARCHAR2) IS
    BEGIN
        dbms_alert.signal('email_notification', message);
        END send_alert;
END alerts_pkg;
```

[Team LiB]