CHAPTER 12 Create, Alter, and Drop Tables

Oracle SQL By Example, Fourth Edition by Alice Rischert. Published by Prentice Hall. Copyright © 2008 by Pearson Education, Inc.

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- Creating and Dropping Tables
- Altering Tables and Manipulating Constraints

This chapter introduces you to the Data Definition Language (DDL) commands associated with tables, the type of database object most frequently used. Table 12.1 provides an overview of other commonly used object types discussed in this and the following chapters.

TABLE 12.1 Commonly Used Database Object Types

DATABASE OBJECT	PURPOSE	FIND MORE INFORMATION
Table	Stores data	This chapter
View	Used for security and to hide complexity	Chapter 13, “Indexes, Sequences, and Views”
Index	Improves data access speed	Chapter 13
Sequence	Generates unique key values	Chapter 13
Synonym	Provides an alternative name for a database object	Chapter 15, “Security”
Directory	Points to a directory location outside the Oracle database	This chapter and Chapter 11, “Insert, Update, and Delete”
STORED DATABASE OBJECTS CREATED USING PL/SQL		

Trigger	Individual PL/SQL program that executes on DML operations
Function	Program that returns a single value
Procedure	Program may return zero, one, or many values
Package	Collection of procedures, functions, or other PL/SQL constructs bundled together

The DDL commands allow you to create, modify, and remove database objects. This chapter discusses the options available with respect to tables, which allow you to manipulate column definitions and constraints. Because database constraints enforce business rules and data integrity, understanding constraints such as the primary key, foreign key, check, and unique constraints is essential to learning about a relational database.

Keep in mind that a DDL statement automatically issues an implicit COMMIT.

LAB 12.1 Creating and Dropping Tables

LAB OBJECTIVES

After this lab, you will be able to:

- Create and Drop Tables
- Create Constraints

For your overall comprehension of the SQL language, it is very helpful to learn about the DDL involved in the creation of a physical table and its associated constraints. This lab provides an overview of naming conventions, syntax, and constraint considerations. While this chapter concentrates primarily on the SQL command and syntax options, you will see later in the lab that if you use SQL Developer, you do not need to remember the precise syntax of all the SQL commands. You can select commands using a simple menu click instead. However, you must know that choosing a menu option does not eliminate the need to understand SQL and the effects of the executed actions. Furthermore, not all the various command syntax options are available in SQL Developer.

Creating Tables

You create tables by using the CREATE TABLE command, in either of two ways. The first method is to specify the columns and their data types explicitly; the second method is to create a table based on an existing table.

The following statement creates a table called TOY that consists of four columns. A NOT NULL constraint is specified for the DESCRIPTION column. The newly created table contains no data.

```
CREATE TABLE toy
   (toy_id              NUMBER(10),
   description              VARCHAR2(
15) NOT NULL,
   last_purchase_date  DATE,
   remaining_quantity  NUMBER(6))
```

When defining a table and columns, you need to know about the naming restrictions and conventions.

TABLE NAMES

A table name must be unique within a database schema; no other database object, such as another table, view, or synonym, can have the same name. Every database object name must be no longer than 30 characters; cannot include spaces or hyphens, but can have underscores; and must begin with a letter. A table name should describe the nature of the data contained in the table; for consistency, choose either singular or plural names for all your tables.

COLUMN NAMES

A column name must be unique within a table and should not exceed 30 characters. It should be descriptive of the values stored in the column. You can document the meaning of individual columns or tables in more detail with the COMMENT command, discussed later in this chapter.

A column is defined not only by a name but also by data type and length, where appropriate. When creating multiple columns, use a comma to separate the column definitions.

By default, table and column names are stored in the Oracle database in all uppercase. It is possible to create table names and column names with mixed case, special characters, and spaces if you use double quotes around the table and column names, but doing so defies the conventions used by most Oracle database installations.

Many corporations have created their own standard column and naming conventions. Compliance with naming standards simplifies the task of identifying database objects for developers. Furthermore, it shortens the learning curve for individuals involved in the maintenance and support of a system.

Be consistent with your table and column names in terms of abbreviations and the use of either singular or plural names.

To simplify the understanding of relationships among the tables, use the identical column name for both the primary and foreign key columns whenever possible. For example, the STUDENT_ID foreign key column in the ENROLLMENT table references the primary key column of the same name in the STUDENT table.

TABLE CREATION SYNTAX

Following is the simplified syntax of a CREATE TABLE statement. (There are many more syntax options; only the most frequently used syntax choices are listed here.)

```
CREATE [GLOBAL TEMPORARY] TABLE
tablename
   (columnname data_type [DEFAULT
expr]|
   [GENERATED ALWAYS AS (expr)
VIRTUAL]
   [column_constraint_clause]
       [, columnname data_type
[DEFAULT expr]|
          [GENERATED ALWAYS AS
(expr) VIRTUAL]
             [column_constraint_cla
use]...]
   [table_constraint_clause]
   Creating and Dropping Tables
```

```
   )
   [physical_storage_clause]
   [ENABLE|DISABLE ROW MOVEMENT]
   [temporary_table_clause]
   [AS query]
```

The CREATE TABLE syntax shows that you must list the individual column name and the respective data type; the default expression, virtual column expression, and a column constraint clause are optional. The column constraint clause has a number of individual syntax options that allow you to restrict the values in an individual column. Because a table actually doesn't contain just one column, the syntax shows that the various syntax portions, consisting of column name, data type, default expression, virtual column expression, and column constraint clause, may be repeated for each subsequent column.

Besides an individual column constraint, a table may have table constraints that restrict one or multiple columns. Tables require physical storage, with individual storage parameters defined in the storage clause. As previously mentioned, you can create a table based on another table; you do this by using the AS QUERY clause. You can create a temporary table with the GLOBAL TEMPORARY keywords and the use of temporary_table_clause.

As you work your way through this lab, you will learn about all the different clauses and gain an understanding about the core functionality of the CREATE TABLE command.

Commonly Used Oracle Data Types

Based on the nature of the type of data you want to store, you choose the appropriate data type. This section reviews Oracle's most commonly used data types. You can find more details in Appendix I, "Oracle Data Types."

CHARACTER DATA

Character data is stored in columns of data type VARCHAR2, CHAR, CLOB, or LONG. When creating or altering a table, the VARCHAR2 and CHAR data types require a column length. The maximum length of a VARCHAR2 column is 4,000 bytes or 4,000 characters, depending on the syntax. (Only for multilingual databases is it important to distinguish between bytes and characters because a single character can take more than one byte. The Oracle default column definition is bytes.)

A fixed-length CHAR column has a maximum length of 2,000. A name such as Smith stored in the LAST_NAME column defined as VARCHAR2(25) stores only 6 characters versus 25 characters in fixed-length CHAR(25)-defined column because the CHAR adds trailing spaces.

The CLOB data type stores up to 4 GB of data, and a table may have multiple CLOB columns. You might still see the use of the LONG data type. It stores up to 2 GB of data in a single column; only one LONG column per table is allowed, and you cannot use character functions on a LONG column. Oracle recommends the use of the CLOB data type instead of the LONG data type because the LONG data type is used only for backward compatibility.

NUMERIC DATA

The format of the NUMBER data type is *NUMBER(p,s)*, where *p* is the *precision* (or total number of digits) and *s* is the *scale* (the number of digits to the right of the decimal point). The NUMBER data type can store up to 38 decimal digits of precision. The definition of NUMBER(5,2) on a column allows you to store values between –999.99 and 999.99. A number such as 1,000 is rejected, and a value such as 80.999 is rounded up to 81.00. Use the NUMBER data type for data on which you need to calculate, not for phone numbers or zip codes. For example, in the STUDENT schema, the ZIP column of the ZIPCODE table is stored as a VARCHAR2 rather than a NUMBER data type because it requires leading zeros.

The BINARY_FLOAT and BINARY_DOUBLE data types store floating-point numbers in 32-bit and 64-bit format. These data types are particularly useful if you require complex and/or fast arithmetic computations.

Floating-point numbers do not have a scale because the number of digits to the right of the decimal point is not restricted. Floating-point numbers can have a decimal anywhere from the first to the last digit, or they can have none at all.

DATE AND TIME DATA

The DATE data type stores the century, year, month, day, hour, minute, and second. It has its own internal format, which can be displayed using different format masks. You can store dates from January 1, 4712 B.C. to December 31, 9999 A.D. The TIMESTAMP data type includes additional fractional seconds, and the TIMESTAMP WITH TIME ZONE data type enables you to keep track of time across geographic regions. The TIMESTAMP WITH LOCAL TIME ZONE is concerned with the date and time in the local region only. INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND handle differences between dates and times.

BINARY DATA AND LARGE OBJECT DATA TYPES

Oracle allows you to save binary data such as images, audio, and video in data types called BLOB, RAW, LONG RAW, and BFILE. A BFILE data type points to a binary operating system file.

Integrity Constraints

When creating tables, you typically create them with integrity constraints. These constraints enforce the business rules of a system. For instance, "The salary of an employee may not be a negative number" may be enforced with a check constraint on the salary column, or "An employee must have a unique Social Security number" can be enforced with a NOT NULL constraint and a unique constraint. Constraints ensure data integrity and data consistency among all applications, no matter which program. They ease the burden of programming the business rules in individual applications because the database enforces the constraint.

The following CREATE TABLE statement creates a table called TAB1 with several types of constraints.

```
CREATE TABLE tab1
   (col1 NUMBER(10)   PRIMARY KEY,
   col2 NUMBER(4)     NOT NULL,
   col3 VARCHAR2(5)   REFERENCES
zipcode(zip)
   ON DELETE CASCADE,
```

508
509

```
   col4 DATE           DEFAULT
SYSDATE,
   col5 VARCHAR2(20)   UNIQUE,
   col6
NUMBER              CHECK(col6 <
100))
```

THE PRIMARY KEY CONSTRAINT

The first column of the table, COL1, has a PRIMARY KEY constraint, also referred to as an *entity integrity constraint*. The primary key ensures that all values in this column are NOT NULL and are unique. This is enforced through a unique index automatically created by Oracle, unless an index already exists. (Indexes are discussed in Chapter 13.) When the table TAB1 is created, Oracle automatically generates a name for this constraint, which looks something like this: SYS_C0030291. This constraint name is not terribly meaningful because it does not identify the table for which the constraint was created or the constraint type. You'll learn how to name constraints shortly.

Every table usually has one primary key, consisting of one or more columns. The combination of all values in a multicolumn primary key, also called a concatenated primary key, must also be unique. Primary keys should be static, which means no updates are usually performed. The primary key values are typically created by a number-generating sequence. This type of key is also referred to as an *artificial*, or *surrogate*, key and has the advantage that these values are completely meaningless and therefore not subject to updates. For primary keys, the NUMBER data type is a better choice than the VARCHAR2 data type because it is not prone to punctuation, case-sensitivity, and spelling mistakes, which make it more difficult to distinguish whether two records are identical. A table without a primary key should have at least a unique constraint.

UNIQUE CONSTRAINTS

To enforce unique values on an individual or a group of columns, you create a unique constraint for a table. In this example, column COL5 has a UNIQUE constraint. Before determining the primary key, there are often alternate keys that are candidates for the primary key.

Phone numbers and Social Security numbers are examples of alternate keys with unique constraints. However, these keys are often not chosen as primary keys because they may allow null values, or the values may be subject to updates. These types of business keys are very useful for end users querying the data, and often uniqueness is enforced through the unique constraint.

The most distinguishing characteristic between a primary key constraint and a unique constraint is that a unique constraint allows null values.

FOREIGN KEY CONSTRAINTS

A foreign key constraint, also referred to as a *referential integrity constraint*, ensures that the values in the foreign key correspond to values of a primary key. The column COL3 contains a FOREIGN KEY constraint. The keyword REFERENCES, followed by the ZIPCODE table and the ZIP column in the ZIPCODE table in parentheses, indicate that COL3 is a foreign key 509
510 to the ZIP column of the ZIPCODE table. The FOREIGN KEY constraint indicates the domain of values for COL3; in other words, the only valid values for the COL3 column are zip codes found in the ZIP column of the ZIPCODE table and null values. Following is an excerpt from the previous CREATE TABLE statement, which shows the relevant foreign key constraint syntax.

```
CREATE TABLE tab1
...
```

```
    col3 VARCHAR2(5) REFERENCES
 zipcode(zip) ON DELETE CASCADE,
 ...
```

Alternatively, the foreign key can be created with this syntax; it does not mention the ZIP column. It is simply assumed that it is the primary key of the referenced table.

```
 col3 VARCHAR2(5) REFERENCES zipcode
    ON DELETE CASCADE,
```

When you're defining a FOREIGN KEY constraint on a table, the column name does not have to be identical to the column name it references. For example, COL3 is the foreign key name, and ZIP is the referencing column name, but the data type and length must agree. Foreign keys almost always reference primary keys, but they may reference unique constraints. Foreign keys should usually be indexed; you will learn more about their purpose and syntax in Chapter 13.

DELETES AND THE FOREIGN KEY

By default the foreign key constraint is of type DELETE RESTRICT; in effect, parent rows cannot be deleted if child rows exist. An ON DELETE CASCADE clause indicates that when a parent row is deleted, the corresponding row or rows in this child table will be deleted as well. In the previous SQL statement, DELETE CASCADE is explicitly specified, so if a row in the ZIPCODE table is deleted, any rows with the same zip code are deleted from the TAB1 table.

Another possible clause for defining the delete behavior of the foreign key is the clause ON DELETE SET NULL. A delete of a zip code will update the corresponding child rows in TAB1 to null, provided that the COL3 column allows null values.

RECURSIVE RELATIONSHIPS

A *recursive relationship* is also known as a self-referencing relationship; the PREREQUISITE and COURSE_NO columns of the COURSE table provide an example where a foreign key references the primary key constraint of the same table. A recursive relationship is enforced just like any other foreign key; you will see an example how to create such a relationship later in this chapter.

CHECK CONSTRAINTS

Check constraints enforce logical expressions on columns, which must evaluate to true for every row in the table. The COL6 column has a CHECK constraint that constrains the column to values less than 100. A null value is allowed, as the column does not have a NOT NULL constraint.

```
CREATE TABLE tab1
...
   col6 NUMBER CHECK(col6 < 100))
...
```

The following is another example of a check constraint; this constraint on a column called STATE restricts the values to the states listed in the IN clause.

```
state VARCHAR2(20) CHECK(state IN
   ('NY', 'NJ', 'CT', 'FL', 'CA'))
```

NOT NULL CHECK CONSTRAINTS

The column COL2 contains a check constraint you are already familiar with: NOT NULL. Any insertions or changes to data that change the values in this column to NULL are rejected.

```
CREATE TABLE tab1
...
   col2 NUMBER(4) NOT NULL,
...
```

A check constraint can also be written as follows, but the previous form is simpler.

```
col2 NUMBER(4) CHECK (col2 IS NOT
NULL),
```

You define the NOT NULL constraints for columns that must always contain a value. For example, the LAST_NAME column of the INSTRUCTOR table is defined as a NOT NULL column, and therefore you cannot create or update a row in the INSTRUCTOR table unless a value exists in the column.

THE DEFAULT COLUMN OPTION

The column COL4 specifies a DEFAULT option, which is not a constraint. When a row is inserted into TAB1 and no value is supplied for COL4, SYSDATE is inserted by default.

```
CREATE TABLE tab1
...
   col4 DATE DEFAULT SYSDATE,
...
```

In an INSERT statement, the keyword DEFAULT explicitly specifies the default value. If a column is not listed in the INSERT statement and a default column value defined, then the default value is automatically inserted in the table. In an UPDATE statement the DEFAULT keyword resets a column value to the default value. Refer to Lab 12.2 for more examples.

A default value can be created for any column except the column or columns of the primary key. Often you choose a default value that represents a typical value. You can combine a default value with a NOT NULL constraint to avoid null values in columns. For example, if the typical COST of a course is 1095, you might want to create such a default value for this column. Another effect of default values and the NOT NULL constraint is that if you want to retrieve costs that are less than 1595 or null, you don't have to write the following query.

```
SELECT *
   FROM course
WHERE NVL(cost,0) < 1595
```

Instead, you simplify the query to the following statement. In the exercises in Lab 12.2, you'll learn more about the factors to take into consideration when defining columns as null versus not null.

```
WHERE cost < 1595
```

Naming Constraints

Applying names to all constraints is a good habit to adopt; it simplifies identifying constraint errors and avoids confusion and further research. Following is an example of how to name constraints in a CREATE TABLE statement.

```
CREATE TABLE tab1
  (col1 NUMBER(10),
  col2 NUMBER(4) CONSTRAINT
tab1_col2_nn NOT NULL,
  col3 VARCHAR2(5),
  col4 DATE DEFAULT SYSDATE,
  col5 VARCHAR2(20),
  col6 NUMBER,
  CONSTRAINT tab1_pk PRIMARY
KEY(col1),
  CONSTRAINT tab1_zipcode_fk
FOREIGN KEY(col3)
  REFERENCES zipcode(zip),
  CONSTRAINT tab1_col5_col6_uk
UNIQUE(col5, col6),
  CONSTRAINT tab1_col6_ck
CHECK(col6 < 100),
  CONSTRAINT tab1_col2_col6_ck
CHECK(col2 > 100 AND col6 >20))
```

```
Table created.
```

Some of the constraint names are next to each column; these are *column-level constraints*. The constraint names at the end of the statement are *table-level constraints*. Constraint names cannot exceed 30 characters and must be unique within the user's schema. In this example, the constraint names consist of the name of the table and column (or an abbreviated version) and a two-letter abbreviation that identifies the type of constraint.

Ideally, you follow a standard naming convention, determined by your organization. In this book, the convention for naming primary key constraints is to use the name of the table plus the _PK suffix. The foreign key constraint contains the abbreviated name of the child table, then the parent table and the _FK suffix. The unique constraint lists the table name and the columns plus the _UK suffix. Often you must abbreviate table and column names; otherwise, you exceed the 30-character constraint name limit. The CHECK constraint called TAB1_COL6_CK, contains the table name and column name plus the _CK suffix.

All the examples listed here show the constraints added at the time of table creation. In Lab 12.2 you will see how to add constraints after the table exists.

It is best to name constraints explicitly, for clarity and to manipulate them more easily, as shown in Lab 12.2. Also, when a SQL statement, such as an INSERT, an UPDATE, or a DELETE statement, violates a constraint, Oracle returns an error message with the name of the constraint, making it easy to identify and understand the source of the error.

Table-Level and Column-Level Constraints

Constraints are defined on two possible levels—either on the column level or on the table level. A column-level constraint refers to a single column and is defined together with the column. A table-level constraint references one or multiple columns and is defined separately, after the definition of all the columns. Column-level constraints are also referred to as *inline constraints*, and table-level constraints are called *out-of-line constraints*.

All constraints except for the NOT NULL constraint can be defined at the table level. You must use a table-level constraint if you are constraining more than one column.

The general syntax for the column constraint clause is listed as follows: It shows the NOT NULL, PRIMARY, FOREIGN, UNIQUE, and CHECK constraint options.

```
CONSTRAINT constraintname]
[NULL|NOT NULL] |
[REFERENCES tablename [(columname)]
   [ON DELETE {CASCADE|ON DELETE
SET NULL}] |
[[UNIQUE|PRIMARY KEY]
   [USING INDEX
   [(CREATE INDEX indexname
   ON tablename
(columnname[,columname...])]
   [storage_clause])]] |
[CHECK (check_condition)]
[ENABLE|DISABLE]
[VALIDATE|NOVALIDATE]
[NOT DEFERRABLE|DEFERRABLE]
[INITIALLY IMMEDIATE|INITIALLY
DEFERRED]
```

The constraint name is optional and must be preceded with the keyword CONSTRAINT. Unless you specify otherwise, your column allows nulls; the underline indicates that this is the default. The foreign key constraint is defined with the REFERENCES keyword; it has two choices with regard to DELETEs, as indicated with the vertical bar, or pipe symbol (|). One is the ON DELETE CASCADE keyword, the other is ON DELETE SET NULL. If you don't list either of these two choices, the deletion of rows is restricted—that is, your deletion is successful only if no child rows exist.

Because unique and primary key constraints automatically create a unique index, you can use an optional index clause to explicitly create an index with predefined storage parameters. This allows you to define the index on a different tablespace (which is often on a different physical device) for better performance. 513

514 The next constraint option is the check constraint syntax. You see that check_condition is within a set of parentheses. All constraints can be either disabled or enabled (the default). The VALIDATE and NOVALIDATE options indicate whether the constraint is enforced for existing and new data or only for subsequently created data.

The DEFERRABLE clauses enforce the timing when the constraint is checked. The default is to check the constraint when the data manipulation occurs. If a constraint is set as DEFERRABLE, the constraint is not checked immediately but only when the transaction is committed.

The table-level constraint is listed after the column definitions. The syntax is as follows.

```
[CONSTRAINT constraintname]
   [UNIQUE
(columnname[,columname...])|
   PRIMARY KEY
(columnname[,columname...])]
   [USING INDEX
   [CREATE INDEX indexname ON
tablename
(columnname[,columname...])]
[storage_clause]] |
[FOREIGN KEY
(columnname[,columname...])]
   REFERENCES tablename
[(columname[,columname...])]
   [ON DELETE {CASCADE|ON DELETE
SET NULL}] |
[CHECK (check_condition)]
[ENABLE|DISABLE]
[VALIDATE|NOVALIDATE]
[NOT DEFERRABLE|DEFERRABLE]
```

```
[INITIALLY IMMEDIATE|INITIALLY
DEFERRED]
```

Enforcing Business Rules with Constraints

Constraints enforce rules and procedures based on rules established within an organization. For example, a rule that a student must have a last name is enforced through a NOT NULL constraint. Another rule may state that students must live in a valid zip code, and this rule can be imposed with a referential integrity constraint referencing the ZIPCODE table and a NOT NULL constraint on the ZIP column of the STUDENT table. You can apply a check constraint to make sure course costs fall within a certain range. The data type of a column determines what kind of data is allowed for entry and perhaps the maximum length. A unique constraint prevents duplicate entry of Social Security numbers into an EMPLOYEE table. A data consistency rule may state that for any deletion of a student record, all corresponding enrollment and grade records are deleted; you do this with a referential integrity foreign key constraint and the ON DELETE CASCADE keyword.

Other business rules may not be as easily enforceable with any of Oracle's declarative constraints. For instance, your rule might state that a student cannot enroll after a class has already started. To enforce this rule, you have to ensure that the value in the ENROLL_DATE column of the ENROLLMENT table contains a value less than or equal to the value in the START_DATE_TIME column of the SECTION table for the student's enrolled section. Database triggers enforce such rules and fire on the INSERT, UPDATE, or DELETE operation of a specific table and check other tables to see if the values satisfy the business rule criteria. If they do not, the operation will fail, and the statement will be rejected.

DATABASE TRIGGERS

Database triggers are PL/SQL programs associated with a table, view, system, or database event. The following trigger is used to audit data modification. The trigger fires before the UPDATE of each row on the STUDENT table, and it automatically updates the MODIFIED_DATE column with the SYSDATE function, filling in the current date and time whenever any update in the table takes place.

```
CREATE OR REPLACE TRIGGER
student_trg_bur BEFORE UPDATE ON
STUDENT
```

```
FOR EACH ROW
BEGIN
   :new.modified_date:=SYSDATE;
END;
/
```

This database trigger is written in Oracle's PL/SQL language and you will learn more about PL/SQL and triggers in general in *Oracle PL/SQL by Example*, 4th edition, by Benjamin Rosenzweig and Elena Silvestrova Rakhimov (Prentice Hall, 2008).

Without going into great depth about the language, you can see that the trigger has the name STUDENT_TRG_BUR, and it fires before the update of an individual row in the STUDENT table. The BEFORE keyword indicates that the trigger can access the new value before it is applied and can change the value with the :NEW.column name correlation value. The value in the MODIFIED_DATE column is changed upon the UPDATE of the affected rows to the current date and time, as indicated by the SYSDATE function.

Triggers are useful for filling in primary key values from sequences, and Chapter 13 provides an example of this. Triggers can also enforce referential integrity constraints rather than apply a foreign key constraint.

However, it is preferable to use Oracle's built-in declarative constraints, such as a foreign key constraint, to enforce these rules. Constraints are easier to maintain, and using them is simpler and faster than duplicating identical functionality in a trigger.

WHERE TO ENFORCE BUSINESS RULES

You can enforce business rules either on the client side through the front-end program or on the database server. Alternatively, the business logic can also reside on a third tier, perhaps an application server. At times you might see that some rules are enforced in multiple places. The decision often depends on a number of factors: Rules imposed across all applications are often done on the database server, because it enforces rules consistently without the need to change and code the logic in many programs.

On the other hand, certain data validation needs to be performed in the front-end program. For example, if a business rule states that a salary must be larger than zero and not null, you may perform this validation within the data entry screen. If the rule is violated, the user receives a friendly error message to correct the data entry. Otherwise, it is annoying to the user to enter the data, only to find out the database rejected the entry. If the salary can be updated by programs other than the front-end screen, you might consider enforcing the rules on both the client front-end program and the server. Be sure to keep the rules consistent throughout.

There are many options to keep in mind regarding the placement of business rules when you are designing applications and database systems, including considerations about user-friendliness, data integrity, consistency, future maintenance, and elimination of duplicate efforts on both the front end and the back end.

Comprehensive data validation is one of the keys to any successful database operation, and finding the right balance requires a thorough understanding of many aspects of a system. Ignorance of data validation leads to invalid data, data inconsistencies, formatting problems, programming and processing errors, and misinterpretation of data.

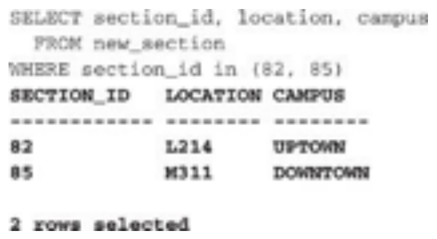
The Virtual Column Option

Oracle 11*g* added the ability to define virtual columns on a table. A virtual column can be derived from the other columns of the table, be the result of a function, or be a constant expression. There is no physical data stored in the column, and therefore no DML operations are permitted against the column.

A virtual column is helpful because it can simplify queries, and you can create indexes on it. The following example shows the creation of the CAMPUS column on the NEW_SECTION table.

```
CREATE TABLE new_section
(section_id NUMBER(8) NOT NULL,
   course_no NUMBER(8) NOT NULL,
   section_no NUMBER(3) NOT NULL,
   start_date_time DATE,
   location VARCHAR2(50),
   campus VARCHAR2(20)
   GENERATED ALWAYS AS
   (CASE WHEN SUBSTR(location,1,1)=
'L'
THEN 'UPTOWN'
ELSE 'DOWNTOWN' END) VIRTUAL)
```

Depending on the location of the section, the values in the virtual column show the appropriate CAMPUS, based on the evaluation of the LOCATION expression.



```
SELECT section_id, location, campus
  FROM new_section
WHERE section_id in (82, 85)
SECTION_ID   LOCATION CAMPUS
------------ -------- --------
82           L214     UPTOWN
85           M311     DOWNTOWN

2 rows selected
```

Any DML that attempts to alter the virtual column’s value is not permitted.

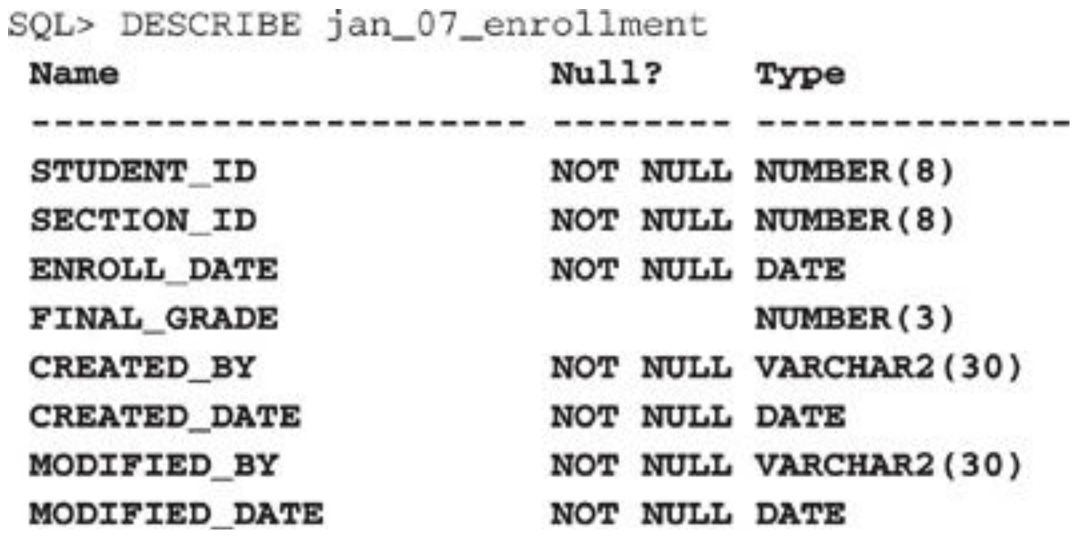
```
UPDATE new_section
   SET campus = ‘UPTOWN’
UPDATE section
    *
ERROR at line 1:
ORA-54017: UPDATE operation
disallowed on virtual columns
```

Creating Tables Based on Other Tables

One method of creating a table is to base it on another table or tables, using a query construct. You can choose to include the data or not. The following example creates a table called JAN_07_ENROLLMENT, based on the January 2007 enrollment rows in the ENROLLMENT table.

```
 CREATE TABLE jan_07_enrollment AS
 SELECT *
   FROM enrollment
 WHERE enroll_date >=
TO_DATE(‘01/01/2007’, ‘MM/DD/YYYY’)
   AND enroll_date <
TO_DATE(‘02/01/2007’, ‘MM/DD/YYYY’)
 Table created.
```

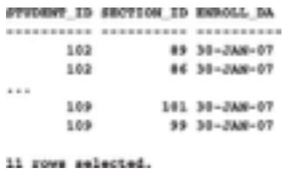
The database feedback message “Table created” confirms that the JAN_07_ENROLLMENT table is successfully created. You can see the columns and their data types by using the SQL*Plus DESCRIBE command.



```
SQL> DESCRIBE jan_07_enrollment
 Name                      Null?    Type
 ------------------------- -------- -------------
 STUDENT_ID                NOT NULL NUMBER(8)
 SECTION_ID                NOT NULL NUMBER(8)
 ENROLL_DATE               NOT NULL DATE
 FINAL_GRADE                        NUMBER(3)
 CREATED_BY                NOT NULL VARCHAR2(30)
 CREATED_DATE              NOT NULL DATE
 MODIFIED_BY               NOT NULL VARCHAR2(30)
 MODIFIED_DATE             NOT NULL DATE
```

The new table has the same columns, data types, and lengths as the ENROLLMENT table on which it is based. A SELECT statement on the new table confirms that the inserted data is equal to the condition listed in the WHERE clause.

```
SELECT student_id, section_id,
enroll_date
   FROM jan_07_enrollment
```



```
STUDENT_ID SECTION_ID ENROLL_DA
---------- ---------- ---------
       102         89 30-JAN-07
       102         86 30-JAN-07
...
       109        101 30-JAN-07
       109         99 30-JAN-07

11 rows selected.
```

You can use the same syntax to create a table without data. Instead of the WHERE clause restricting specific rows from the ENROLLMENT table, here no rows are returned. The ROWNUM pseudocolumn indicates the order in which Oracle selects rows from a table or set of tables. The first selected row has ROWNUM 1, the second has ROWNUM 2, and so on. Because the query asks for less than one row, the statement subsequently creates an empty table.

```
CREATE TABLE jan_07_enrollment AS
SELECT *
   FROM enrollment
  WHERE rownum < 1
```

Alternatively, you can also write the statement with a query that never evaluates to true, such as in the next example.

```
CREATE TABLE jan_07_enrollment AS
SELECT *
   FROM enrollment
  WHERE 1 = 2
```

Tables created with this syntax construct do not inherit the primary key, foreign keys, constraints, indexes, column default values, or any other objects associated with the base table except the NOT NULL constraints, which receive a system-generated name that starts with SYS_.

If a SELECT statement in a CREATE TABLE statement joins two tables or more, it is best not to use the asterisk wildcard in the SELECT list. The tables being joined may contain columns with the same name, and you get an error message when Oracle attempts to create two columns with the same name in one table.

Renaming Tables

You can rename tables by using the RENAME command. The syntax of the command is as follows.

```
RENAME oldname TO newname
```

You can use the RENAME command to rename not just tables but also views and synonyms; these object types are discussed in the following chapters.

The following statement renames the JAN_07_ENROLLMENT table JAN_07.

```
RENAME jan_07_enrollment TO jan_07
```

```
Table renamed.
```

Alternatively, you can execute the ALTER TABLE command, which is discussed in Lab 12.2, or use SQL Developer's Rename menu option.

```
ALTER TABLE jan_07_enrollment
RENAME TO jan_07
```

Constraint names and dependent database objects, such as indexes and triggers, are not renamed when the table name is changed. Any privileges on the table that have been granted to other users remain intact. Dependent objects such as views become invalid and need to be recompiled.

Dropping Tables

You can drop tables when they are no longer needed by using the DROP TABLE command, whose syntax is a follows.

```
DROP TABLE tablename [CASCADE
CONSTRAINTS] [PURGE]
```

When you drop a table, the table and its data are removed, along with any indexes, triggers, and constraints.

```
DROP TABLE jan_07
```

```
Table dropped.
```

Starting with Oracle 10*g*, the table is moved into a recycle bin from which it can be recovered; this is also referred to as *flashback drop*. Prior Oracle versions permanently removed the table and reclaimed the space from the database. If you do not want to place the table in the recycle bin, use the PURGE syntax option.

Other tables may be dependent on the dropped table as a domain for a foreign key reference. For example, if you drop the ZIPCODE table, an Oracle error message occurs because there are other tables with a foreign key referencing the ZIP column of the ZIPCODE table. One solution is to disable or drop the individual foreign key constraints with individual ALTER TABLE commands on these dependent tables, which you will learn about in Lab 12.2. Another is to let Oracle drop the foreign key constraints with the CASCADE CONSTRAINTS option.

Do not actually execute the following statement unless you are prepared to reload the data from the ZIPCODE table and add the foreign key constraints on the STUDENT and INSTRUCTOR tables.

```
DROP TABLE zipcode CASCADE
CONSTRAINTS
```

When you use the DROP TABLE command, database objects that depend on the table, such as a view referencing the table, synonyms, or PL/SQL packages, procedures, and functions, become invalid. To find out which objects reference a table, query the data dictionary view ALL_ DEPENDENCIES or USER_DEPENDENCIES or click on SQL Developer's Dependencies tab for a given object. If any rights on the table were granted to other users (for example privileges to SELECT, INSERT, UPDATE, or DELETE), they are removed. If you re-create the table with the same name and want other users to continue having these privileges, you need to reissue the privileges (see Chapter 15).

Restoring a Dropped Table

You learned about some of the capabilities of the FLASHBACK TABLE command in Chapter 11. The FLASHBACK TABLE command allows you to restore a dropped table from the recycle bin. The syntax of the FLASHBACK TABLE command is repeated here.

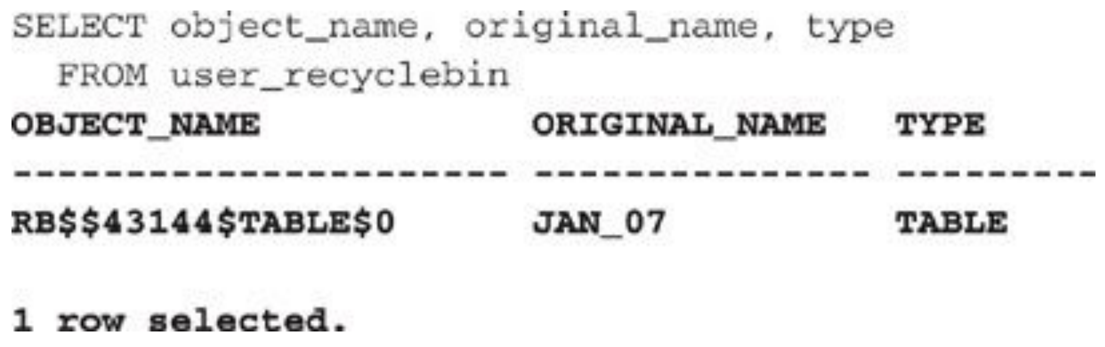
```
FLASHBACK TABLE table name [,
tablename...] TO
```

```
{{SCN|TIMESTAMP} expr [ENABLE|
DISABLE TRIGGERS]|
   BEFORE DROP [RENAME TO
newtablename]}
```

The following statements show how the table JAN_07 is dropped and subsequently restored.

```
DROP TABLE jan_07
Table dropped.
```

Data associated with the dropped table is stored in the recycle bin. You can query the USER_RECYCLEBIN data dictionary table or its synonym RECYCLEBIN.



```
SELECT object_name, original_name, type
  FROM user_recyclebin
OBJECT_NAME            ORIGINAL_NAME   TYPE
---------------------- --------------- ---------
RB$$43144$TABLE$0      JAN_07          TABLE

1 row selected.
```

The following command restores the dropped table. You can refer to the table by either the original name or the system-generated recycle bin name.

```
FLASHBACK TABLE jan_07 TO BEFORE
DROP
Flashback complete.
```

If any triggers, constraints, or indexes are associated with the table, they are restored as well, except for bitmap join indexes and referential integrity constraints to other tables. All these objects have their recycle bin names, not the original name. Before you issue the FLASHBACK TABLE command, make a note of the names so you can rename them to their original names.

PURGING THE RECYCLE BIN

You can use the PURGE command to purge an individual table or index, or the entire recycle bin. The syntax is as follows.

```
PURGE {{TABLE|INDEX}
recyclebin_objectname|
   RECYCLEBIN|DBA_RECYCLEBIN|
   TABLESPACE tablespacename [USER
user]}
```

The following command reclaims all the space in the user’s recycle bin.

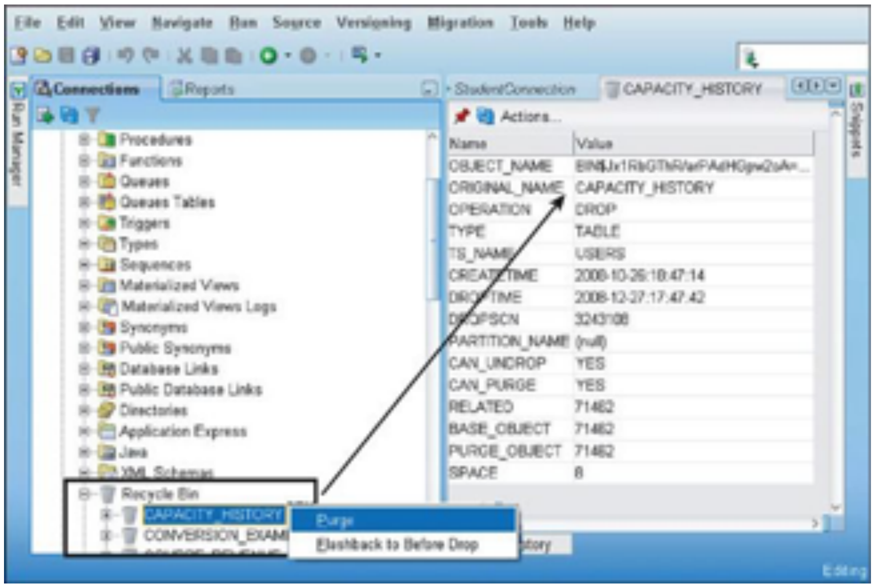
```
PURGE RECYCLEBIN
Recyclebin purged.
```

A database administrator (DBA) has the additional option of reclaiming all objects from the systemwide DBA_RECYCLEBIN or can use the TABLESPACE clause of the PURGE command to reclaim space in a specific tablespace for a specific user account. You will learn more about table-spaces shortly.

THE RECYCLE BIN AND SQL DEVELOPER

You can view the recycle bin from SQL Developer (see Figure 12.1). Clicking on one of the objects in the recycle bin provides you with the details similar to the USER_RECYCLEBIN data dictionary table. When you right-click the table, you get two menu options: You can either purge the table or flash back to the state before the drop.

FIGURE 12.1 Recycle bin in SQL Developer



Name	Value
OBJECT_NAME	BIN$Jx1RbGThR/arFAdHGpw2sA=...
ORIGINAL_NAME	CAPACITY_HISTORY
OPERATION	DROP
TYPE	TABLE
TS_NAME	USERS
CREATETIME	2008-10-26:18:47:14
DROPTIME	2008-12-27:17:47:42
DROPSCN	3243108
PARTITION_NAME	(null)
CAN_UNDROP	YES
CAN_PURGE	YES
RELATED	71462
BASE_OBJECT	71462
PURGE_OBJECT	71462
SPACE	8

TRUNCATE TABLE Versus DROP TABLE

The TRUNCATE TABLE command, discussed in Chapter 11, removes all data from the table; however, the structure of the table remains intact, as do any triggers and grants. Like the DROP TABLE command, it does not generate any rollback information and does not fire any triggers, should they exist on the table. The TRUNCATE statement is a DDL command and implicitly issues a COMMIT. By default, the TRUNCATE TABLE command deallocates all of the table's storage except for the initial extent(s); you can retain all the existing storage extents with the REUSE STORAGE clause.

```
TRUNCATE TABLE grade REUSE STORAGE
```

The Storage Clause

A CREATE TABLE statement may have an optional storage clause that specifies space definition attributes. Each table allocates an initial extent that defines how much diskspace is reserved for the table at the time of creation. After the table runs out of the initial extent, Oracle automatically allocates additional space, based on the storage parameters of the NEXT extent parameter.

The following statement creates a table called CTX_BOOKMARK with a storage clause specifying an initial size of 5 MB on the tablespace called USERS. Once the table is out of the allocated space, each subsequent extent allotted is 1 MB, as indicated with the NEXT parameter.

```
CREATE TABLE ctx_bookmark
   (bookmark_id NUMBER,
   container_id NUMBER,
   bookmark_tx VARCHAR2(300) NULL,
   modified_date DATE)
   TABLESPACE users
   STORAGE (INITIAL 5M NEXT 1M)
   PCTFREE 20
```

A tablespace consists of one or more physical data files. For performance reasons, tables and indexes are usually stored in separate tablespaces, located on different physical disk drives. To find out which tablespaces are available to you, query the data dictionary view USER_TABLE-SPACES.

If no specific storage parameters are defined, the default storage parameters of the tablespace apply. Statements with a missing tablespace name create the data on the default tablespace assigned when the user was created and listed in the USER_USERS data dictionary view. If a user account does not have any rights to create any table objects, or no rights on certain tablespaces, these rights must be granted to the user first. You will learn more about granting access to table-spaces in Chapter 15. 522

523

ESTIMATING THE TABLE SIZE

Estimating the size of a table is useful to reduce the amount of wasted space. When you create a table, you can pre-allocate space with the INITIAL syntax parameter and plan for any subsequent expansion with the NEXT parameter option.

While determining how much space will actually be consumed by the table is an inexact science; you can estimate how much initial space to allocate. You determine a rough size by entering sample data in the table and then computing the statistics (discussed in Chapter 18, "SQL Optimization"). This way you can update the data dictionary information with statistics about the table, including the average row length in bytes (the AVG_ROW_LEN column in the USER_TABLES data dictionary view).

You multiply the average row length figure by the number of rows you expect in the table plus about 10 to

15 percent for overhead. Increase the number by how much free space you want to leave in each data block for updates that increase the size of the rows; you determine this figure as a percentage with the PCTFREE ("percent free") parameter in the storage clause.

```
avg_row_len in bytes * number of
rows * (1 + PCTFREE/100) * 1.15
```

Next, you see the most frequently used options of the storage clause. The PCTUSED parameter determines when a block becomes available again for insertions after its used space falls below the PCTUSED integer.

```
[TABLESPACE tablespacename]
[PCTFREE integer]
[PCTUSED integer]
[COMPRESS|NOCOMPRESS]
[STORAGE
   ([INITIAL integer [K|M]]
   [NEXT integer [K|M]])]
```

To determine the total allocated space of an existing table, you query the BYTES column in the data dictionary view USER_SEGMENTS or DBA_SEGMENTS.

The SQL command syntax in this book highlights the most relevant syntax options. Oracle's SQL commands often include a myriad of different options, some of which are rarely used and therefore not included here. If you need to look up the complete syntax in the Oracle documentation, refer to Appendix G, "Navigating the Oracle Documentation."

PARTITIONING TABLES

Partitioning a table essentially means splitting a table into smaller pieces. The individual partitions often are stored in different tablespaces. Very large tables become more manageable for database administration tasks when they are stored on different partitions. Partitioning can also improve performance because input/output (I/O) can be balanced. Partitioning can be accomplished in many different ways, and Oracle offers a variety of 523
syntax options. For example, you 524 can place sales data by year in individual partitions. There is no need to change any SQL statements for DML operations because the partitioning is completely transparent. If you want to access an individual partition specifically

with a DML statement, Oracle provides syntax to do so. You most often use partitioning if the tables are very large, such as tables in the gigabyte range and tables containing hundreds of millions of rows.

DATA COMPRESSION

Oracle enables you to use data compression to save storage space; this can be desirable for large database tables in data warehouses where very few updates and deletes take place.

Oracle's Other Table Types

The vast majority of data is stored in an "ordinary" Oracle table. The following paragraphs list other table types (temporary tables, index-organized tables, and external tables) for completeness only; a detailed discussion of these table types goes beyond the scope of this book.

Oracle allows object-oriented capabilities with tables; in practice, however, object-oriented database table features are only slowly gaining acceptance.

TEMPORARY TABLES

When a query becomes too complicated, you can resolve it by writing part of the data to a temporary table before continuing with the main query. Oracle allows two types of temporary tables: session-specific and transaction-specific. The data in a temporary table is visible to multiple sessions or transactions, but only with respect to the data created by each session or transaction. When the session or transaction is complete, the data is deleted.

A temporary table is session-specific when created with the ON COMMIT PRESERVE ROWS keywords and transaction specific when created with the ON COMMIT DELETE ROWS keywords.

The following statement creates a session-specific temporary table that will retain its value until the session ends, not when a transaction ends because of an issued COMMIT or ROLLBACK command.

```
CREATE GLOBAL TEMPORARY TABLE
s_num_rows
   (student_id NUMBER,
   last_name VARCHAR2(25),
   num_classes_enrolled NUMBER)
```

```
ON COMMIT PRESERVE ROWS
```

You enter values into the table with an INSERT statement.

```
INSERT INTO s_num_rows
VALUES (123, ‘Hesse’, 5)
```

The next temporary table is transaction-specific, as you can tell from the ON COMMIT DELETE ROWS keywords. You use the SELECT command to populate rows to this table.

```
CREATE GLOBAL TEMPORARY TABLE
t_grade
   ON COMMIT DELETE ROWS AS
   SELECT student_id,
AVG(numeric_grade) AS avg_grade
   FROM grade
   WHERE student_id IN (SELECT
student_id FROM enrollment WHERE
final_grade IS NOT NULL)
   GROUP BY student_id
```

Temporary tables behave much like regular tables, whereby you can add indexes, triggers, and some types of constraints. However, certain restrictions apply. For example, no referential integrity constraints are allowed.

```
SQL> DESCRIBE t_grade
 Name                            Null?    Type
 ------------------------------- -------- ---------
 STUDENT_ID                      NOT NULL NUMBER(8)
 AVG_GRADE                                NUMBER
```

When are temporary tables useful? Use temporary tables in cases where doing so simplifies the query logic and the query is infrequently executed. Keep in mind that this may not be the most efficient way to execute the query, but as with all other queries, testing against a representative data set will determine whether a temporary table solves your complicated query dilemma.

Data in temporary tables is not stored permanently; it persists only during a session or transaction; however, the structure of the temporary table exists until it is explicitly dropped with a DROP TABLE command.

INDEX-ORGANIZED TABLES

When the primary key of a table comprises most or all of the columns in a table, you might want to consider storing the data in an index-organized table. An index-organized table is useful for frequently used lookup tables that hold currencies, state abbreviations, or stock prices with their respective dates. This type of table quickly executes queries that are looking for the primary key value. Queries do not require the lookup of a value in the index first and then the corresponding retrieval of the row in the table because all the data is stored only in the index. You cannot disable or drop the primary key of an index-organized table.

```
CREATE TABLE states
   (state_code VARCHAR2(2),
   state_tx VARChAR2(200),
CONSTRAINT state_pk PRIMARY KEY
(state_code))
ORGANIZATION INDEX
```

EXTERNAL TABLES

With the help of external tables, Oracle allows read access to data stored outside the database, such as legacy systems. SELECT statements are issued against external tables, much as with any other table. You cannot insert into or update and delete from an external table, nor can you build an index on an external table.

To define an external table, you describe the individual columns with Oracle data types and how to map to these columns. A data access driver and external table layer perform the necessary transformation. Because external data remains stored outside the database, no backup or recovery capabilities within Oracle are performed. External tables are useful for loading data into the database, but their setup requires the help of a DBA, and some knowledge of Oracle's SQL*Loader bulk-load utility is useful.

The following is a simple example of a flat ASCII sample file that is located on one of the directories of the Oracle database server. The file may have data such as the following, where the values are separated by commas.

```
102,Crocitto,Fred
103,Landry,J.
104,Enison,Laetia
105,Moskowitz,Angel
106,Olvsade,Judith
107,Mierzwa,Catherine
108,Sethi,Judy
109,Walter,Larry
```

You need to create an ORACLE DIRECTORY entry so the database knows where to find the file on an accessible directory and drive. If you use the Windows operating system, you may choose to specify a directory such as C:\GUEST as the directory (or wherever your file is located).

```
CREATE DIRECTORY dir_guest AS 'C:
\GUEST'
Directory created.
```

You must make sure the Oracle database has operating system read and write access to the operating system directory; otherwise, Oracle cannot load the data. To create an ORACLE DIRECTORY entry within the Oracle database, you must have the DBA privilege or the CREATE ANY DIRECTORY system privilege. Log on as a user with DBA rights or the SYSTEM account then issue the following command.

```
GRANT CREATE ANY DIRECTORY TO
student
```

You create the external table based on the contents of the flat file by using the CREATE TABLE command. The table is named STUDENT_EXTERNAL, and it has three columns: STUDENT_ID, LAST_NAME, and FIRST_NAME. The keywords ORGANIZATION EXTERNAL identify that this table is located outside the Oracle database. TYPE indicates the driver used to read the data. DEFAULT DIRECTORY identifies the directory where the file is located. The structure of the flat file is defined by the individual field names that are to be read, as well as how these individual fields are separated from each other. In this example, commas separate the fields. In addition, the LOCATION keyword indicates the name of the file. This temp.lst file must be located in the directory defined as DIR_GUEST, which maps to the server's C:\GUEST directory.

```
CREATE TABLE student_external
   (student_id NUMBER(3),
   last_name VARCHAR2(25),
   first_name VARCHAR2(25))
   ORGANIZATION EXTERNAL
   (TYPE oracle_loader
   DEFAULT DIRECTORY dir_guest
ACCESS PARAMETERS
   (FIELDS TERMINATED BY ‘,’
   (student_id, last_name,
first_name))
   LOCATION (‘temp.lst’))
Table created.
```

There are different types of files; this example shows a comma-separated value (CSV) file. Some files enclose the text fields in double quotation marks; others have fixed lengths, where the starting and ending positions of each column is predetermined.

After you successfully create the table and place the temp.lst file in the appropriate directory, you can retrieve data from the table.

```
SELECT *
  FROM student_external
STUDENT_ID LAST_NAME                 FIRST_NAME
---------- ------------------------- ----------------
       102 Crocitto                  Fred
       103 Landry                    J.
       104 Enison                    Laetia
       105 Moskowitz                 Angel
       106 Olvsade                   Judith
       107 Mierzwa                   Catherine
       108 Sethi                     Judy
       109 Walter                    Larry

8 rows selected.
```

Although you have learned about the different Oracle table types, most times you will deal with ordinary Oracle tables, and the aforementioned table types currently represent the exception to the norm.

Using the SQL Developer Graphical User Interface (GUI)

The SQL Developer GUI allows you to perform many DDL operations without having to remember the syntax. The result of menu actions can generate the desired SQL language commands.

SQL DEVELOPER'S TABLE-RELATED MENU OPTIONS

The table-related menu in SQL Developer's Connections pane has many choices. Figure 12.2 shows the different menu options available when you right-click an individual table. Some of the menu options are described in this chapter; others are discussed in the chapters that follow. Table 12.2 provides an overview of the functionality and purpose of each menu option.

FIGURE 12.2 Table-related menu choices

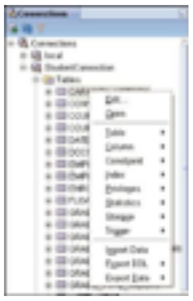


TABLE 12.2 Overview of SQL Developer Table-Related Menu Choices

MENU CHOICE	DESCRIPTION
Edit	Allows you to modify any of the properties related to the table, its columns, constraints, indexes, or storage parameters and more (see Lab 12.2).
Open	Has the same effect as double-clicking the table's node. Displays the various tabs related to showing columns, data, constraints, grants, triggers, flashback data, and so on.
Table	Displays table submenu functions, such as Rename, Copy, Drop, and Truncate. It allows the addition of table comments and counting of the number of rows. 528
Column	529 Allows the addition, renaming, and dropping of columns, as well as the addition of column comments (see Lab 12.2).
Constraint	Performs adding, dropping, renaming, disabling, and enabling of constraints (see Lab 12.2).
Index	Provides functionality to add, drop, and rebuild indexes (see Chapter 13).

Privileges	Grants privileges to and revokes privileges from other users (see Chapter 15).
Statistics	Gathers statistics (see Chapter 18) and validates object (see Chapter 13)
Storage	Is useful for mainly DBAs for managing the storage spaces of objects, including moves between tablespaces and compressing of segments.
Trigger	Allows the creation, dropping, enabling, and disabling of triggers; includes an option to create a trigger to populate the primary key from a sequence.
Import Data	Imports data from an XLS (Microsoft Excel) or CSV (comma-separated value) file into an Oracle table.
Export DDL	Exports the table's DDL to a file, the Clipboard, or the SQL Worksheet window.
Export Data	Exports a table's data into various formats, such as XLS, CSV, SQL*Loader, and INSERT statements.

CREATING A NEW TABLE

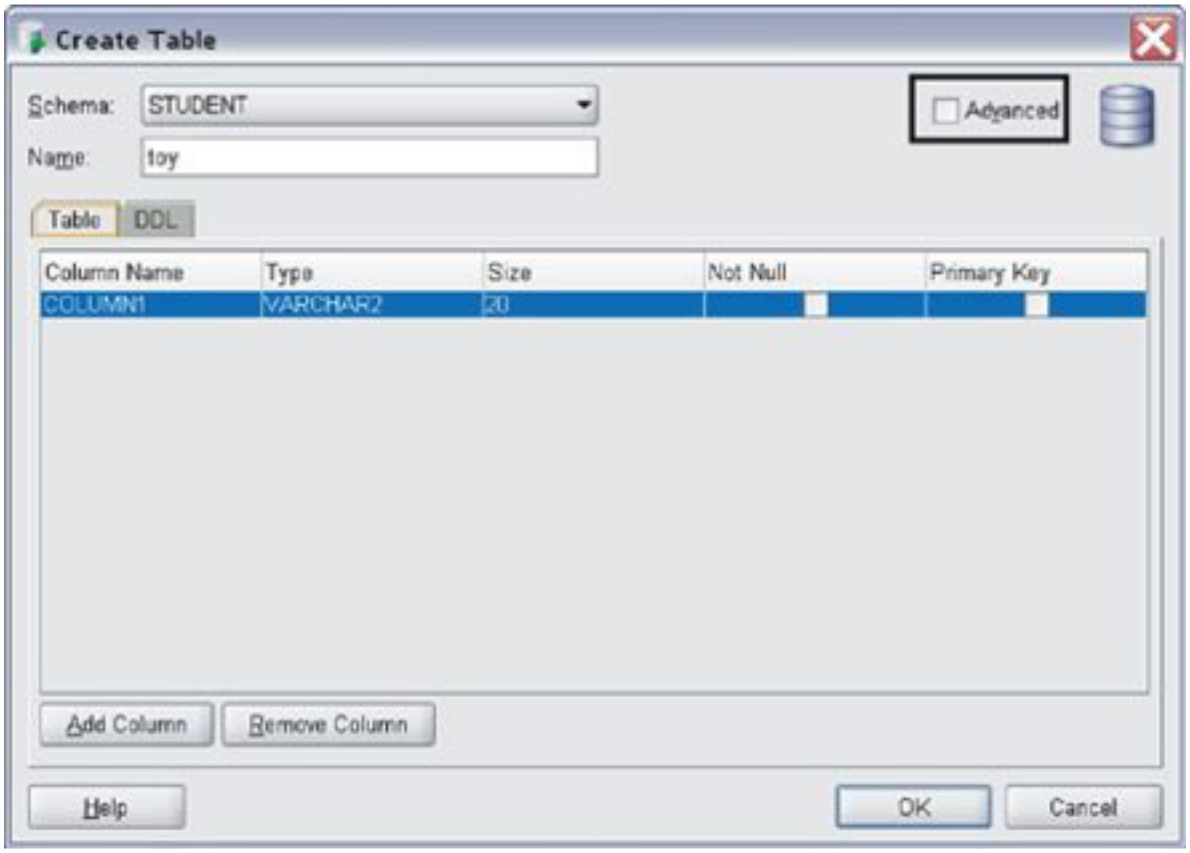
To create a new table, you right-click the Tables node and choose New Table. The dialog box shown in Figure 12.3 appears. You can enter the table name and column names, data type, length, and not null constraint, and you can select a check box to indicate whether this column is part of the primary key. When you click the DDL tab, you can see the DDL command to perform the same functionality.

On the top right of the screen is the Advanced check box, which allows for a wide range of additional syntax options, such as the creation of check constraints, setup of foreign keys, and more.

If you want to create a table based on an existing table and/or change the new table's table or column definitions somewhat, you can extract the DDL for the existing table. This allows you to modify the DDL and create the new table. One of the ways to extract the DDL is by clicking on the SQL tab (see Figure 12.4).

The DDL shows the table name prefixed with the schema name of STUDENT. All the table and column names are in quotation marks to ensure that they are stored in exactly the same case. The last column is followed by the storage clause.

FIGURE 12.3 Create Table dialog box for SQL Developer


Create Table

Schema: STUDENT

Name: toy

[ ] Advanced

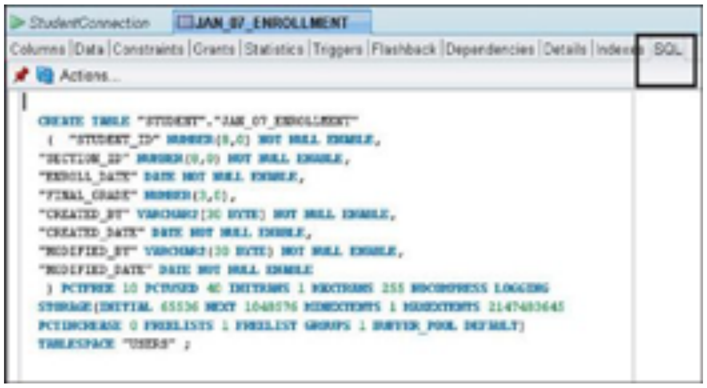
Table | DDL

Column Name	Type	Size	Not Null	Primary Key
COLUMN1	VARCHAR2	20	☐	☐

Add Column | Remove Column

Help | OK | Cancel

FIGURE 12.4 SQL tab showing the DDL for the table


StudentConnection | JAN_07_ENROLLMENT

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL

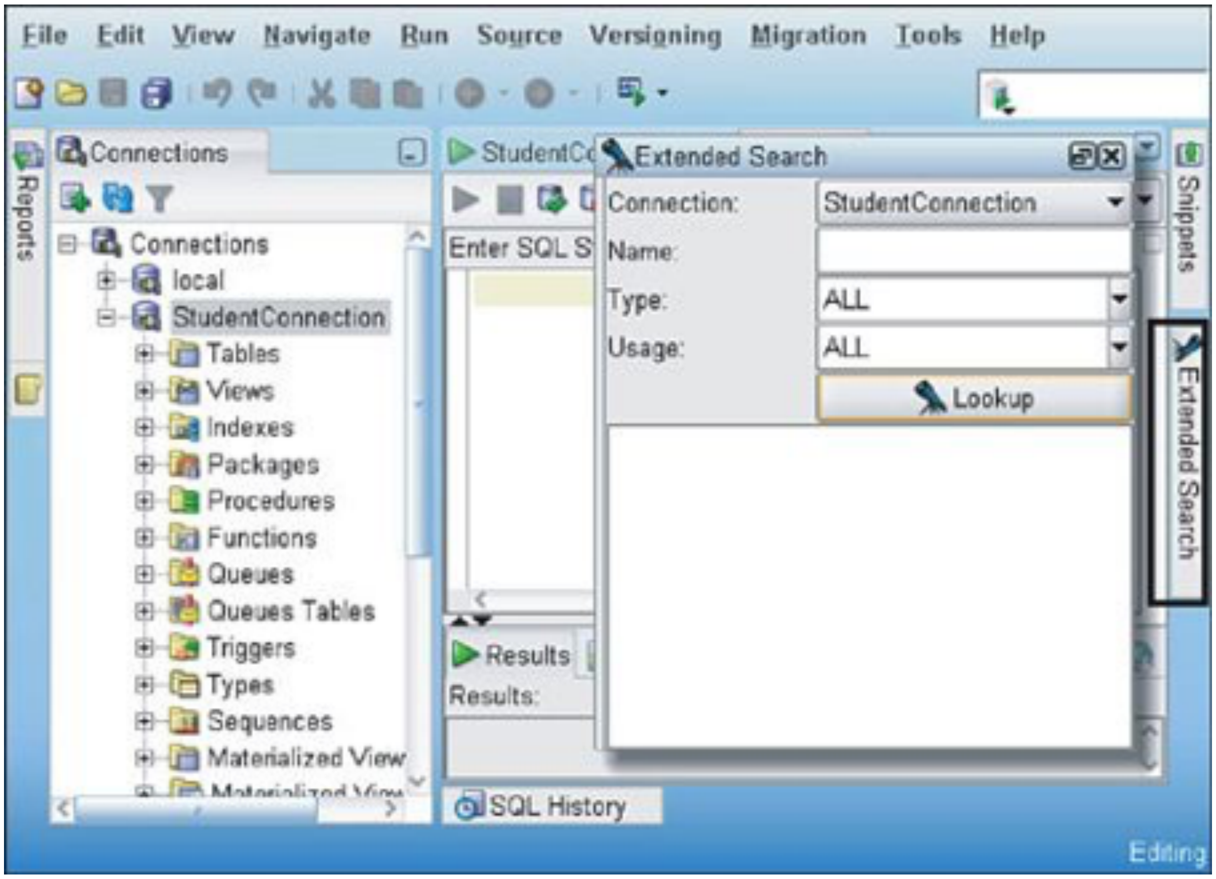
Actions...

```
CREATE TABLE "STUDENT"."JAN_07_ENROLLMENT"
 (  "STUDENT_ID" NUMBER(8,0) NOT NULL ENABLE,
"SECTION_ID" NUMBER(8,0) NOT NULL ENABLE,
"ENROLL_DATE" DATE NOT NULL ENABLE,
"FINAL_GRADE" NUMBER(3,0),
"CREATED_BY" VARCHAR2(30 BYTE) NOT NULL ENABLE,
"CREATED_DATE" DATE NOT NULL ENABLE,
"MODIFIED_BY" VARCHAR2(30 BYTE) NOT NULL ENABLE,
"MODIFIED_DATE" DATE NOT NULL ENABLE
 ) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
TABLESPACE "USERS" ;
```

Using SQL Developer's Extended Search Capabilities

If you want to find any table, index, or column name associated with a specific connection, you can use the Extended Search box to search for the name. The Extended Search box is shown to the right of the SQL Worksheet pane (see Figure 12.5). Alternatively, you can access it by selecting View, Extended Search.

FIGURE 12.5 Extended Search box



Connection: StudentConnection
Name:
Type: ALL
Usage: ALL

LAB 12.1 EXERCISES

a) Explain the error(s) in the following CREATE TABLE statement and rewrite the statement correctly.

```
CREATE TABLE student candidate
   (name       VARCHAR2(25)
   address   VARCHAR2(20)
   city       VARCHAR2
   zip       NUMBER)
```

b) Write and execute a CREATE TABLE statement to create an empty table called NEW_STUDENT that contains the following columns: first name, last name, the description of the first course the student takes, and the date the student registered in the program. Determine the data type and length necessary for each column, based on the tables in the STUDENT schema.

c) Execute the following CREATE TABLE statement and explain the result.

```
CREATE TABLE school_program AS
SELECT last_name||', '||
first_name name
   FROM student
```

```
UNION
SELECT last_name||', '||
first_name
   FROM instructor
```

d) Rename the SCHOOL_PROGRAM table you created in exercise c to SCHOOL_PROGRAM2. Then drop both the SCHOOL_PROGRAM and SCHOOL_PROGRAM2 tables and explain your observations.

e) Execute the following SQL statements to create an empty table called COURSE2 and insert two rows into COURSE2. What do you observe about the values of the COURSE_NO column in the COURSE2 table?

```
CREATE TABLE course2 AS
SELECT *
   FROM course
  WHERE 1 = 2
Table created.
```

```
INSERT INTO course2
   (course_no, description, cost,
prerequisite,
   created_by, created_date,
modified_by, modified_date)
```

```
VALUES
(999, 'Teaching SQL - Part 1',
1495, NULL,
'AMORRISON', SYSDATE,
'AMORRISON', SYSDATE)
```

1 row created.

```
INSERT INTO course2
   (course_no, description, cost,
prerequisite,
   created_by, created_date,
modified_by, modified_date)
VALUES
   (999, 'Teaching SQL - Part 2',
1495, NULL,
   'AMORRISON', SYSDATE,
'AMORRISON', SYSDATE)
```

1 row created.

f) Identify the constraints in the following CREATE TABLE statement and explain their purpose.

```
CREATE TABLE extinct_animal
   (animal_id NUMBER,
   species_id NUMBER,
   name VARCHAR2(30) NOT NULL,
   native_country VARCHAR2(20)
CONSTRAINT
extinct_animal_country_fk
REFERENCES country(country_name),
   remaining NUMBER(2,0),
   CONSTRAINT extinct_animal_pk
PRIMARY KEY(animal_id,
species_id),
   CONSTRAINT
extinct_animal_remaining_ck
CHECK (remaining BETWEEN 0 and
10))
```

g) Rewrite and execute the following CREATE TABLE statement to give the primary key and the foreign key constraints names.

```
CREATE TABLE former_student
   (studid NUMBER(8) PRIMARY KEY,
   first_nm VARCHAR2(25),
   last_nm VARCHAR2(25),
```

532

533

```
   enrolled VARCHAR2(1) DEFAULT
'N',
   zip VARCHAR2(5) REFERENCES
zipcode(zip))
```

h) Rewrite the solution to exercise g to add a UNIQUE constraint on the FIRST_NM and LAST_NM columns.

i) Using the extended search capability of SQL Developer, search for the Student ID column.

LAB 12.1 EXERCISE ANSWERS

a) Explain the error(s) in the following CREATE TABLE statement and rewrite the statement correctly.

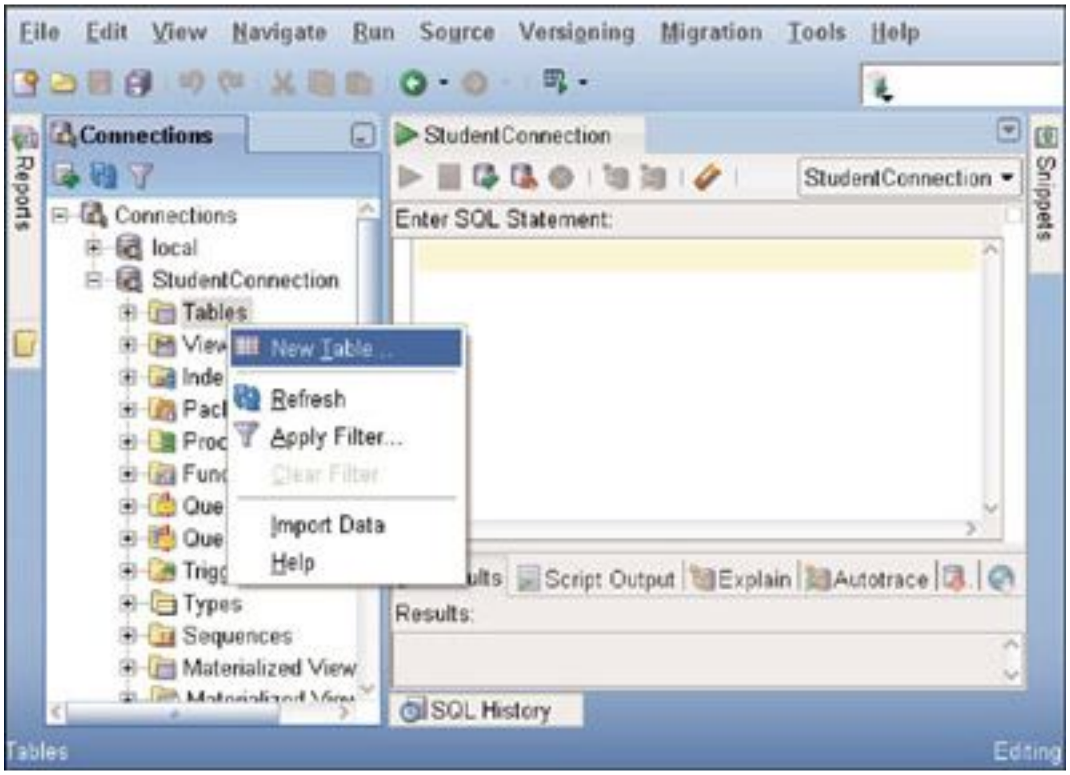
```
CREATE TABLE student candidate
   (name        VARCHAR2(25)
   addres     sVARCHAR2(20)
   city        VARCHAR2
   zip        NUMBER)
```

ANSWER: The statement will not execute because there are three errors. The first error is in the table name; it contains spaces. The next error is the nonexistent length of the CITY column's VARCHAR2 definition. Finally, commas are required to separate the column definitions. Following is the corrected syntax.

```
CREATE TABLE student_candidate
   (name        VARCHAR2(25),
   address    VARCHAR2(20),
   city        VARCHAR2(15),
   zip        NUMBER)
```

If you create the table using SQL Developer, you can avoid some of the simple syntax errors (see Figure 12.6).

FIGURE 12.6 New Table menu



b) Write and execute a CREATE TABLE statement to create an empty table called NEW_STUDENT that contains the following columns: first name, last name, the description of the first course the student takes, and the date the student registered in the program. Determine the data type and length necessary for each column, based on the tables in the STUDENT schema.

ANSWER: The table contains the four columns FIRST_NAME, LAST_NAME, DESCRIPTION, and REGISTRATION_DATE. The first three are of data type VARCHAR2, and REGISTRATION_DATE is of data type DATE.

```
CREATE TABLE new_student
   (first_name          VARCHAR2(2
5),
   last_name           VARCHAR2(25)
,
   description          VARCHAR2(5
0),
   registration_date    DATE)
```

DOCUMENTING THE TABLES AND COLUMNS

When you create tables and columns, you can add comments to them to document their purposes. These comments are stored in the data dictionary for reporting and self-documentation. You can query the involved data dictionary views ALL_COL_COMMENTS and ALL_TAB_COMMENTS or view them in SQL Developer in various menus.

If you want to write SQL to create or modify the comments, the following is an example of how to create a table comment on the NEW_STUDENT table. The comment is enclosed in single quotation marks.

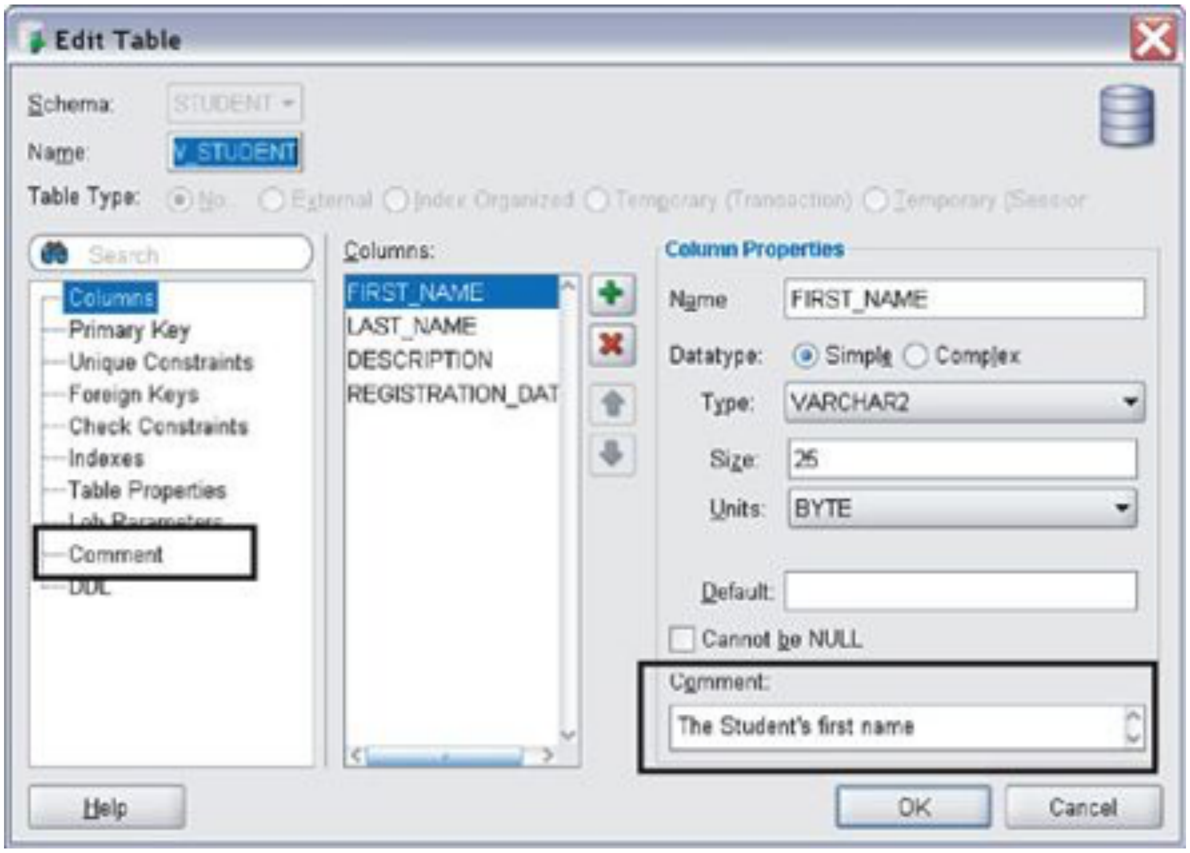
```
COMMENT ON TABLE new_student IS
'Table holding student information
used for exercises'
Comment created.
```

The following is an example of a column comment for the FIRST_NAME column on the NEW_STUDENT table. Two individual quotation marks are necessary to represent a single quote when issuing this statement.

```
COMMENT ON COLUMN
new_student.first_name is 'The
student''s first name.'
Comment created.
```

Figure 12.7 shows where you can edit the table-level and column-level comments within in SQL Developer. This screen is accessed via the Edit Table menu in the Connections pane. The comment box on the bottom right of the screen is for column-level comments. The comment box on the left allows you to edit the comments for the respective table.

FIGURE 12.7 Entry for table and column comments

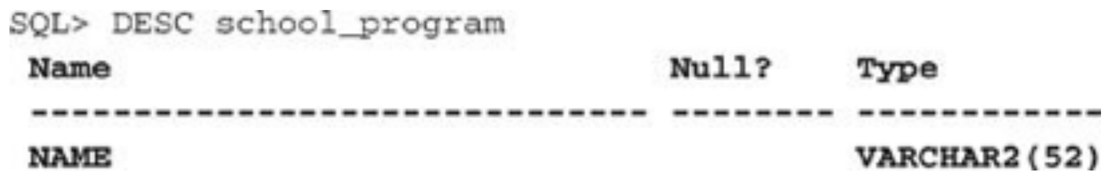


c) Execute the following CREATE TABLE statement and explain the result. CREATE TABLE school_program AS

```
SELECT last_name||‘, ’||first_name
name
   FROM student
UNION
SELECT last_name||‘, ’||first_name
```

```
    FROM instructor
```

ANSWER: The statement creates a table called SCHOOL_PROGRAM, based on a query of two other tables that combines student and instructor names. The first and last names are concatenated into one column.



```
SQL> DESC school_program
 Name                          Null?    Type
 ----------------------------- -------- ------------
 NAME                                   VARCHAR2(52)
```

The Name column in the new table is long enough to accommodate the combined length of the first and last names, plus a comma and a space.

d) Rename the SCHOOL_PROGRAM table you created in exercise c to SCHOOL_PROGRAM2. Then drop both the SCHOOL_PROGRAM and SCHOOL_PROGRAM2 tables and explain your observations.

ANSWER: Use the RENAME and DROP TABLE commands.

The DROP TABLE command fails because the SCHOOL_PROGRAM table no longer exists; it has been renamed SCHOOL_PROGRAM2.

```
RENAME school_program TO
school_program2
```
Table renamed.

```
DROP TABLE school_program
```
DROP TABLE school_program

***ERROR at line 1:**
ORA-00942: table or view does not exist

```
DROP TABLE school_program2
```
Table dropped.

Because the newly created table may not be in your Tables node, refresh the listing to display it. If you use SQL Developer's Rename menu to change the table name, you can look at the DDL tab to see the respective DDL command.

e) Execute the following SQL statements to create an empty table called COURSE2 and insert two rows into COURSE2. What do you observe about the values of the COURSE_NO column in the COURSE2 table?

```
CREATE TABLE course2 AS
SELECT *
   FROM course
```

```
  WHERE 1 = 2
```
Table created.

```
INSERT INTO course2
   (course_no, description, cost,
prerequisite,
   created_by, created_date,
modified_by, modified_date)
VALUES
(999, ‘Teaching SQL - Part 1’,
1495, NULL,
‘AMORRISON’, SYSDATE, ‘AMORRISON’,
SYSDATE)
```
1 row created.

```
INSERT INTO course2
   (course_no, description, cost,
prerequisite,
   created_by, created_date,
modified_by, modified_date)
VALUES
   (999, ‘Teaching SQL - Part 2’,
1495, NULL,
‘AMORRISON’, SYSDATE, ‘AMORRISON’,
SYSDATE)
```
1 row created.

ANSWER: The primary key constraint is not preserved.

When a table is created from another table, constraints are not automatically preserved in the new table, except for the NOT NULL constraints. The COURSE_NO column is the primary key in the COURSE table and, therefore, prevents duplicate values. But when the COURSE2 table is created from the COURSE table, a primary key constraint is not created, so the COURSE_NO column in the COURSE2 table allows duplicate values to be inserted.

536

537

In Lab 12.2, you will learn how to add constraint to an existing table.

f) Identify the constraints in the following CREATE TABLE statement and explain their purpose.

```
CREATE TABLE extinct_animal
(animal_id NUMBER,
species_id NUMBER,
name VARCHAR2(30) NOT NULL,
native_country VARCHAR2(20)
   CONSTRAINT
extinct_animal_country_fk
```

```
   REFERENCES
country(country_name),
remaining NUMBER(2,0),
CONSTRAINT extinct_animal_pk
PRIMARY KEY(animal_id,
species_id),
CONSTRAINT
extinct_animal_remaining_ck
CHECK (remaining BETWEEN 0 AND
10))
```

ANSWER: The first constraint in the EXTINCT_ANIMAL table is a NOT NULL constraint on the NAME column, and because it is not named, it receives a system-generated name. The NATIVE_COUNTRY column is a constraint with a foreign key to values from the COUNTRY_NAME column in a table called COUNTRY, which must exist for the command to be successful.

The concatenated PRIMARY KEY constraint called EXTINCT_ANIMAL_PK consists of the ANIMAL_ID and SPECIES_ID columns. When a primary key on a table contains more than one column, the constraint must be written as a table-level constraint on a separate line of the CREATE

TABLE statement. The CHECK constraint on the column called EXTINCT_ANIMAL_REMAINING_CK checks whether a number inserted or updated is between the values 0 and 10, inclusive.

A NOT NULL constraint on the ANIMAL_ID and SPECIES_ID columns is not required because the columns are defined as the primary key.

g) Rewrite and execute the following CREATE TABLE statement to give the primary key and the foreign key constraints names.

```
CREATE TABLE former_student
   (studid NUMBER(8) PRIMARY KEY,
   first_nm VARCHAR2(25),
   last_nm VARCHAR2(25),
   enrolled VARCHAR2(1) DEFAULT
'N',
   zip VARCHAR2(5) REFERENCES
zipcode(zip))
```

ANSWER: You can write the syntax in various forms. In addition, SQL Developer can perform the same tasks to create the table and accompanying constraints.

Following are some SQL syntax variations. The constraint definitions are moved to the end of the CREATE TABLE statement, where they are created with specific names.

```
CREATE TABLE former_student
   (studid NUMBER(8),
   first_nm VARCHAR2(25),
   last_nm VARCHAR2(25),
   enrolled VARCHAR2(1) DEFAULT
‘N’,
   zip VARCHAR2(5),
   CONSTRAINT former_student_pk
PRIMARY KEY(studid),
   CONSTRAINT
former_student_zipcode_fk FOREIGN
KEY(zip)
REFERENCES zipcode(zip))
```

Alternatively, the following is syntax that uses the column-level constraints.

```
CREATE TABLE former_student
   (studid NUMBER(8) CONSTRAINT
former_student_pk PRIMARY KEY,
   first_nm VARCHAR2(25),
   last_nm VARCHAR2(25),
   enrolled VARCHAR2(1) DEFAULT
'N',
   zip VARCHAR2(5) CONSTRAINT
former_student_zipcode_fk
REFERENCES zipcode(zip))
```

When a constraint is not named and an error occurs, you receive a system-generated constraint error. The following shows such an example.

```
INSERT INTO former_student
(studid, first_nm, last_nm,
enrolled, zip)
VALUES
(101, 'Alex', 'Morrison', NULL,
'10005')
1 row created.
```

```
INSERT INTO former_student
(studid, first_nm, last_nm,
enrolled, zip)
VALUES
(101, 'Alex', 'Morrison', NULL,
'11717')
INSERT INTO former_student
*
ERROR at line 1:
```

```
ORA-00001: unique constraint
(STUDENT.SYS_C001293) violated
```

From this error message, it is impossible to figure out which column(s) caused the error; you can only determine that the constraint is in the STUDENT schema. You need to look up the name of the constraint in the Oracle data dictionary view USER_CONSTRAINTS or ALL_CONSTRAINTS to determine the reason for the error.

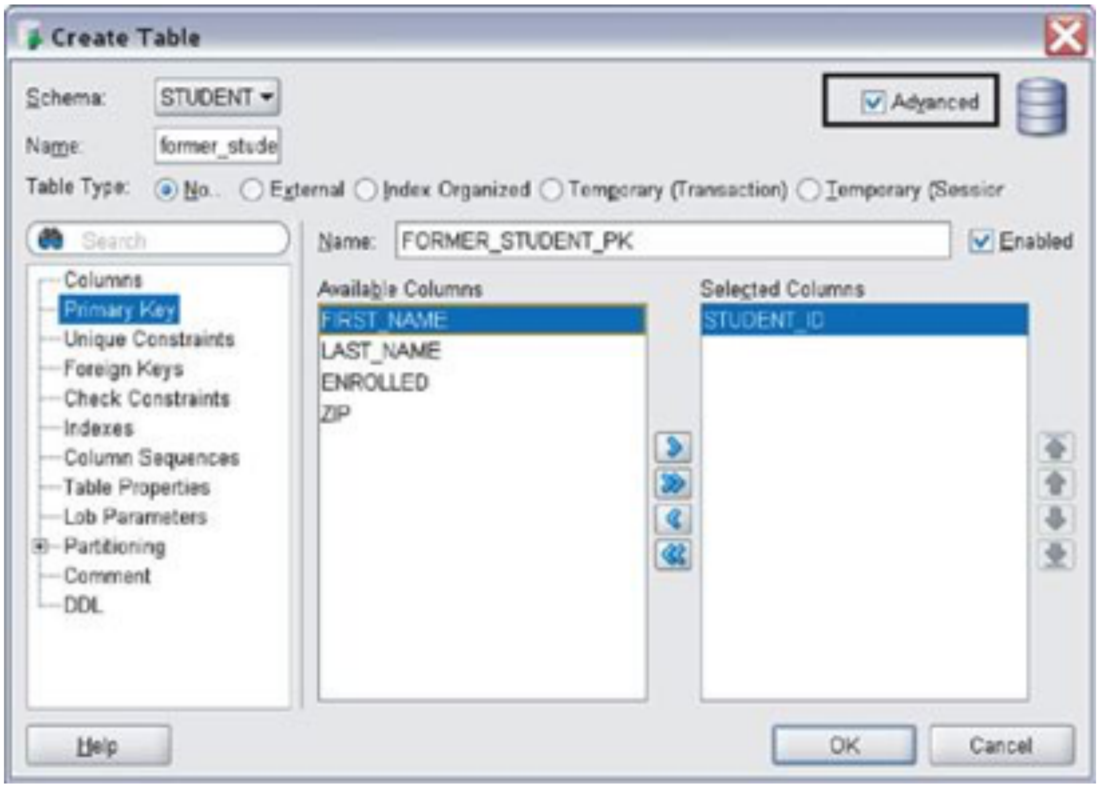
The system-generated name is not informative; therefore, always name your constraints.

In SQL Developer, to create a new table and associated constraints, choose the New Table menu option. You see this menu option when you right-click on the Tables node in the Connection pane. This will bring up the Create Table screen (see Figure 12.8). Note the Advanced check box; when you click on this box, you can add the primary key with the name

FORMER_STUDENT_PK and STUDENT_ID as the selected primary key column.

In the Advanced tab, you can easily add a column default value of 'N' for the ENROLLED column, as shown in Figure 12.9.

FIGURE 12.8 Adding the primary key constraint


Create Table

Schema: STUDENT

Name: former_stude

Table Type: [✓ No...] External Index Organized Temporary (Transaction) Temporary (Session

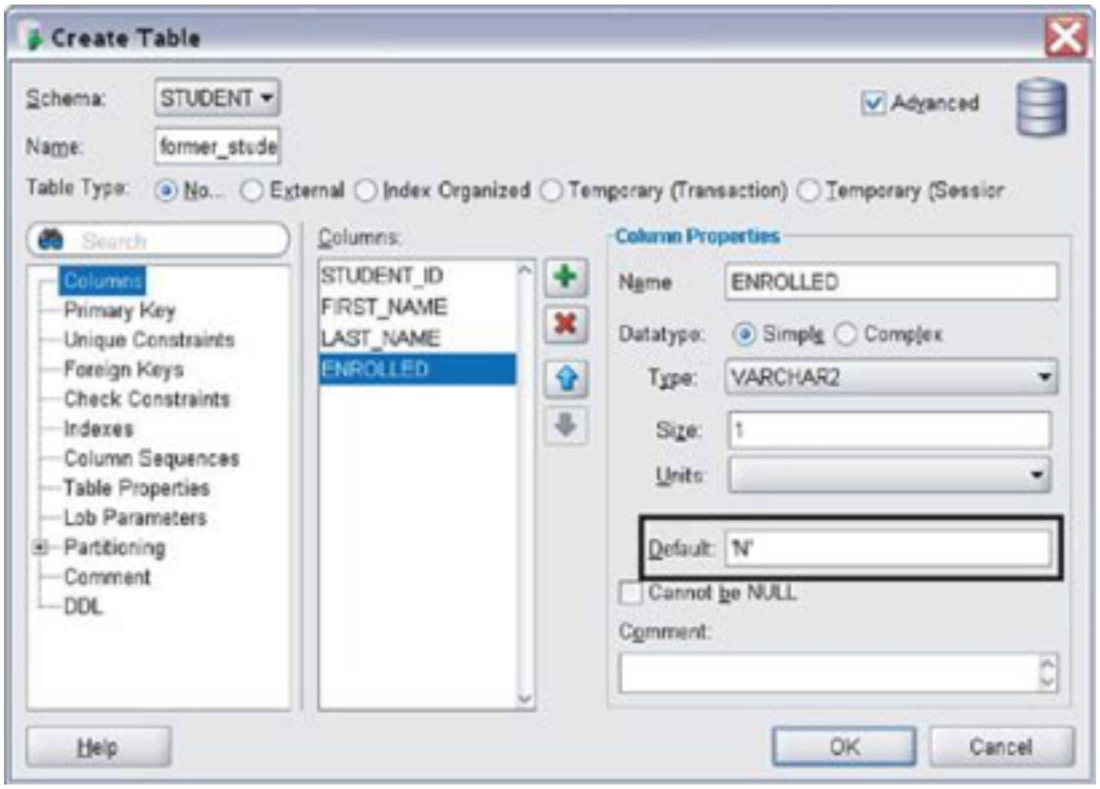
Advanced ✓

Name: FORMER_STUDENT_PK ✓ Enabled

Available Columns: FIRST_NAME, LAST_NAME, ENROLLED, ZIP

Selected Columns: STUDENT_ID

FIGURE 12.9 Adding a default value



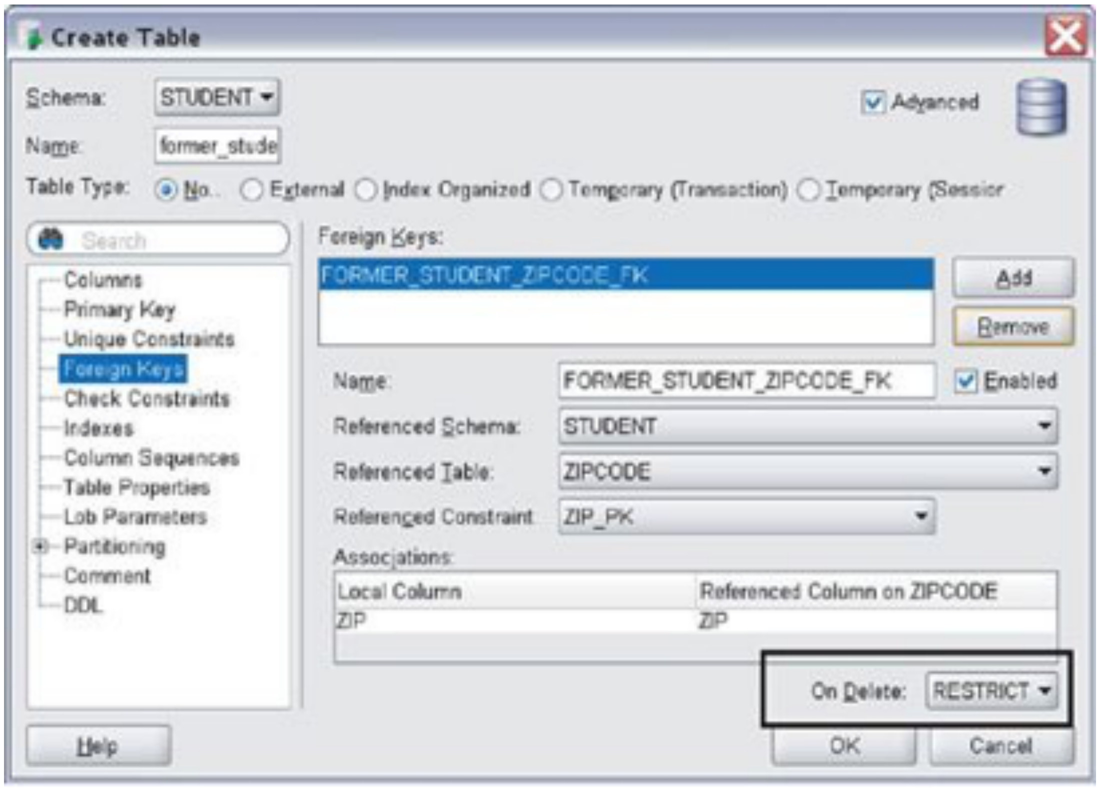
Schema: STUDENT
Name: former_stude
[✓ Advanced]
Table Type: [● No...] ○ External ○ Index Organized ○ Temporary (Transaction) ○ Temporary (Sessior

Columns:
STUDENT_ID
FIRST_NAME
LAST_NAME
ENROLLED

Name: ENROLLED
Datatype: [● Simple] ○ Complex
Type: VARCHAR2
Size: 1
Units:
Default: 'N'
☐ Cannot be NULL
Comment:

Figure 12.10 illustrates the addition of the foreign key constraint on the ZIP column. The foreign key constraint name is suggested, and you can alter it. The foreign key column references the primary key column of the STUDENT schema's ZIPCODE table. The foreign key is a DELETE RESTRICT constraint as shown on the bottom of the screen.

FIGURE 12.10 Adding the foreign key constraint



Schema: STUDENT

Name: former_stude

Table Type: [✓ No...] External Index Organized Temporary (Transaction) Temporary (Session

Foreign Keys: FORMER_STUDENT_ZIPCODE_FK

Name: FORMER_STUDENT_ZIPCODE_FK [✓ Enabled]

Referenced Schema: STUDENT

Referenced Table: ZIPCODE

Referenced Constraint: ZIP_PK

Associations:

Local Column	Referenced Column on ZIPCODE
ZIP	ZIP

On Delete: RESTRICT

h) Rewrite the solution to exercise g to add a UNIQUE constraint on the FIRST_NM and LAST_NM columns.

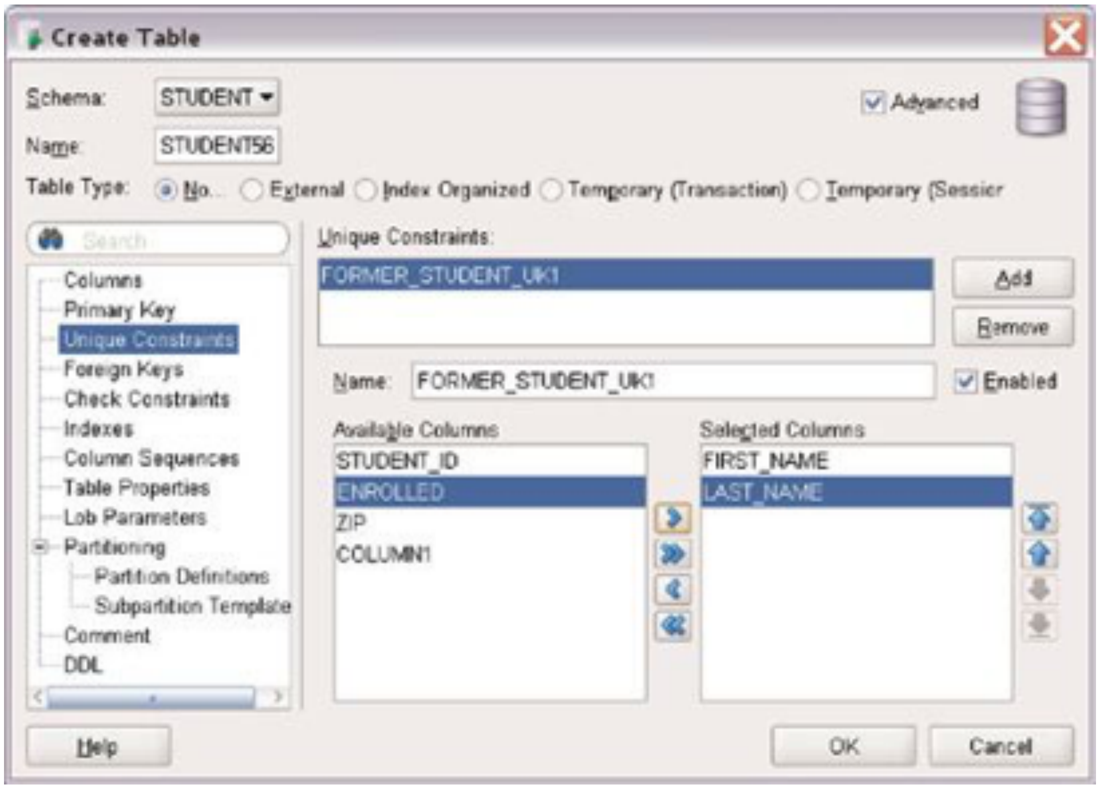
ANSWER: You add the constraint, using a specific name, at the end of the CREATE TABLE statement.

```
CREATE TABLE former_student
    (studid   NUMBER(8),
    first_nm  VARCHAR2(25),
    last_nm   VARCHAR2(25),
    enrolled  VARCHAR2(1) DEFAULT
'N',
    zip      VARCHAR2(5),
    CONSTRAINT former_student_pk
PRIMARY KEY(studid),
    CONSTRAINT
former_student_zipcode_fk FOREIGN
KEY(zip)
    REFERENCES zipcode(zip),
    CONSTRAINT former_student_uk
UNIQUE(first_nm, last_nm))
```

A UNIQUE constraint prevents duplicate values from being inserted into a column. It is different from a PRIMARY KEY constraint because a UNIQUE constraint allows NULL values.

Figure 12.11 shows how to add this in SQL Developer.

FIGURE 12.11 Adding the unique key constraint



Schema: STUDENT

Name: STUDENT56

[✓ Advanced]

Table Type: [✓ No...] External Index Organized Temporary (Transaction) Temporary (Sessior

Unique Constraints: FORMER_STUDENT_UK1

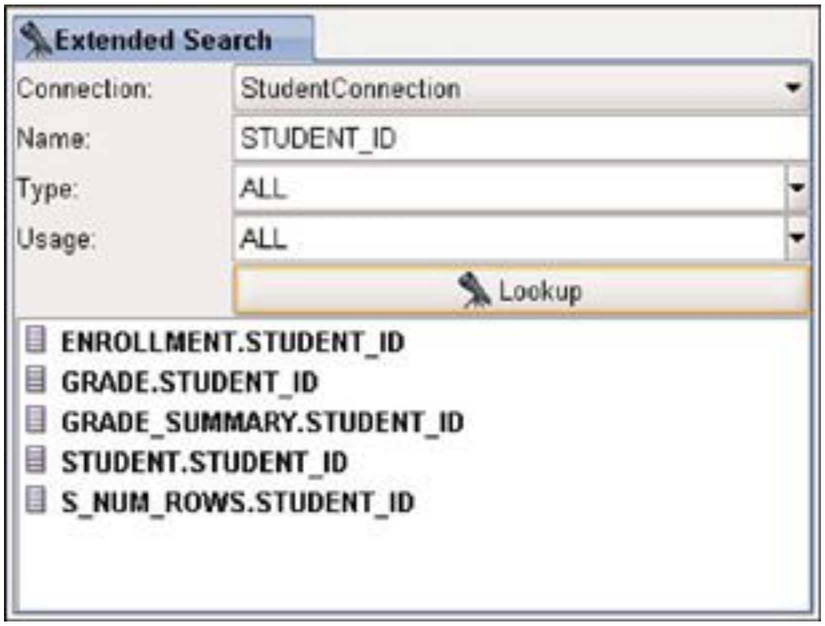
Name: FORMER_STUDENT_UK1 [✓ Enabled]

Available Columns	Selected Columns
STUDENT_ID	FIRST_NAME
ENROLLED	LAST_NAME
ZIP	
COLUMN1	

i) Using the extended search capability of SQL Developer, search for the STUDENT_ID column.

ANSWER: Your results may look similar to Figure 12.12. When you double-click one of the result rows, you can view the respective object. Extended search allows the use of wildcard characters such as % to perform sophisticated searches.

FIGURE 12.12 The Extended Search box with a result



Connection: StudentConnection
Name: STUDENT_ID
Type: ALL
Usage: ALL

ENROLLMENT.STUDENT_ID
GRADE.STUDENT_ID
GRADE_SUMMARY.STUDENT_ID
STUDENT.STUDENT_ID
S_NUM_ROWS.STUDENT_ID

Lab 12.1 Quiz

In order to test your progress, you should be able to answer the following questions.

1) The primary key of the following CREATE TABLE statement is a concatenated primary key.

```
CREATE TABLE class_roster
(class_id NUMBER(3),
   class_name VARCHAR2(20) UNIQUE,
   first_class DATE NOT NULL,
   num_of_students NUMBER(3),
   CONSTRAINT class_roster_pk
   PRIMARY KEY(class_id,
class_name))
```

_______**a)** True

_______**b)** False

2) It is possible to create one table from three different tables in a single CREATE TABLE statement.

_______**a)** True

_______**b)** False

3) The CASCADE CONSTRAINTS keywords in a DROP TABLE statement drop all referencing child tables.

_______**a)** True

_______**b)** False

4) Every column of a table can have one or more constraints.

_______**a)** True

_______**b)** False

5) You cannot create a table from another table if it has no rows.

_______**a)** True

_______**b)** False

6) A CREATE TABLE statement automatically commits all previously issued DML statements.

_______**a)** True

_______**b)** False

7) A foreign key must match a primary key or unique key.

_______**a)** True

_______**b)** False

8) Primary key values should always be subject to frequent change.

_______**a)** True

_______**b)** False

9) The STORAGE clause on a CREATE TABLE statement can specify how much space to allocate.

_______**a)** True

_______**b)** False

10) The data type definitions NUMBER(10) and NUMBER(10,0) are equivalent.

_______**a)** True

_______**b)** False

11) The maximum value for a column defined as NUMBER(3,2) is 999.

_______**a)** True

_______**b)** False

ANSWERS APPEAR IN APPENDIX A.

LAB 12.2 Altering Tables and Manipulating Constraints

LAB OBJECTIVES

After this lab, you will be able to:

- Alter Tables
- Manipulate Constraints

After a table is created, you sometimes need to change its characteristics. The ALTER TABLE command, in conjunction with the ADD, DROP, MODIFY, and

RENAME clauses, allows you to do this. You can add or delete a column; change the length, data type, or default value of a column; or add, drop, enable, disable, or rename a table's integrity constraints.

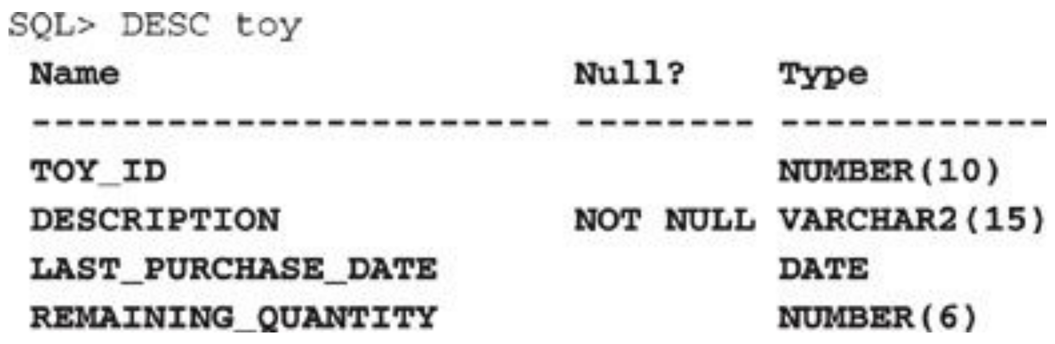
Following is the general syntax for the ALTER TABLE command. You will see examples of these many options throughout this lab and in the following exercises.

```
ALTER TABLE tablename
   [ADD [(columnname
data_type[DEFAULT expr]
   [column_constraint]
   [, columname data_type[DEFAULT
expr]
   [column_constraint]]...)]
   [, table_constraint [,
table_constraint...]]
   [MODIFY [(columname data_type
[DEFAULT expr]
   [column_constraint]
   [MODIFY CONSTRAINT
constraint_name
   [ENABLE|DISABLE] [NOVALIDATE|
VALIDATE]]
   [DROP CONSTRAINT
constraint_name|
   PRIMARY KEY|
   UNIQUE
(columnname[,columname...])
   [CASCADE]
   [DISABLE|ENABLE [VALIDATE|
NOVALIDATE]
   CONSTRAINT constraint_name|
   PRIMARY KEY|
   UNIQUE
(columnname[,columname]...)
   [USING INDEX indexname
[storage_clause]]
   [CASCADE] [{KEEP|DROP}INDEX]]
   [NOT DEFERRABLE|DEFERRABLE]
```

```
[INITIALLY IMMEDIATE|INITIALLY
DEFERRED]
   [RENAME CONSTRAINT
constraint_name TO
new_constraint_name
   [DROP (columnname)|DROP COLUMN
(columnname[,columname...])]
   [SET UNUSED COLUMN columnname|
SET UNUSED
(columnname[,columname...])]
   [DROP UNUSED COLUMNS]
   [RENAME COLUMN columnname TO
newcolumnname]
   [RENAME TO newtablename]
   [storage_clause]
```

Adding Columns

This section takes a look at the columns of the TOY table created at the beginning of Lab 12.1. Following is the SQL*Plus DESCRIBE command to display the structure of the table.

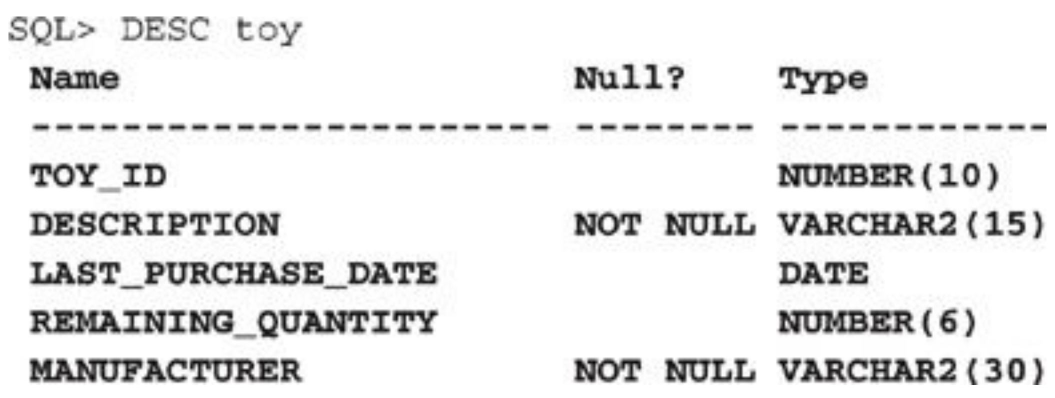


```
SQL> DESC toy
 Name                            Null?    Type
 ------------------------------- -------- ------------
 TOY_ID                                   NUMBER(10)
 DESCRIPTION                     NOT NULL VARCHAR2(15)
 LAST_PURCHASE_DATE                       DATE
 REMAINING_QUANTITY                       NUMBER(6)
```

The following statement alters the TOY table to add a new column called MANUFACTURER.

```
ALTER TABLE toy
   ADD (manufacturer VARCHAR2(30)
NOT NULL)
```
`Table altered.`

The “Table altered” response indicates the successful completion of the operation. When the column is added, it is defined as VARCHAR2(30). The column also has a NOT NULL constraint. When you issue another DESCRIBE command, you see the new column.



```
SQL> DESC toy
 Name                      Null?    Type
 ------------------------- -------- ------------
 TOY_ID                             NUMBER(10)
 DESCRIPTION               NOT NULL VARCHAR2(15)
 LAST_PURCHASE_DATE                 DATE
 REMAINING_QUANTITY                 NUMBER(6)
 MANUFACTURER              NOT NULL VARCHAR2(30)
```

Alternatively, you can add the column and name the constraint as in the following example.

```
ALTER TABLE TOY
   ADD (manufacturer VARCHAR2(30)
CONSTRAINT toy_manufacturer_nn NOT
NULL)
```

545

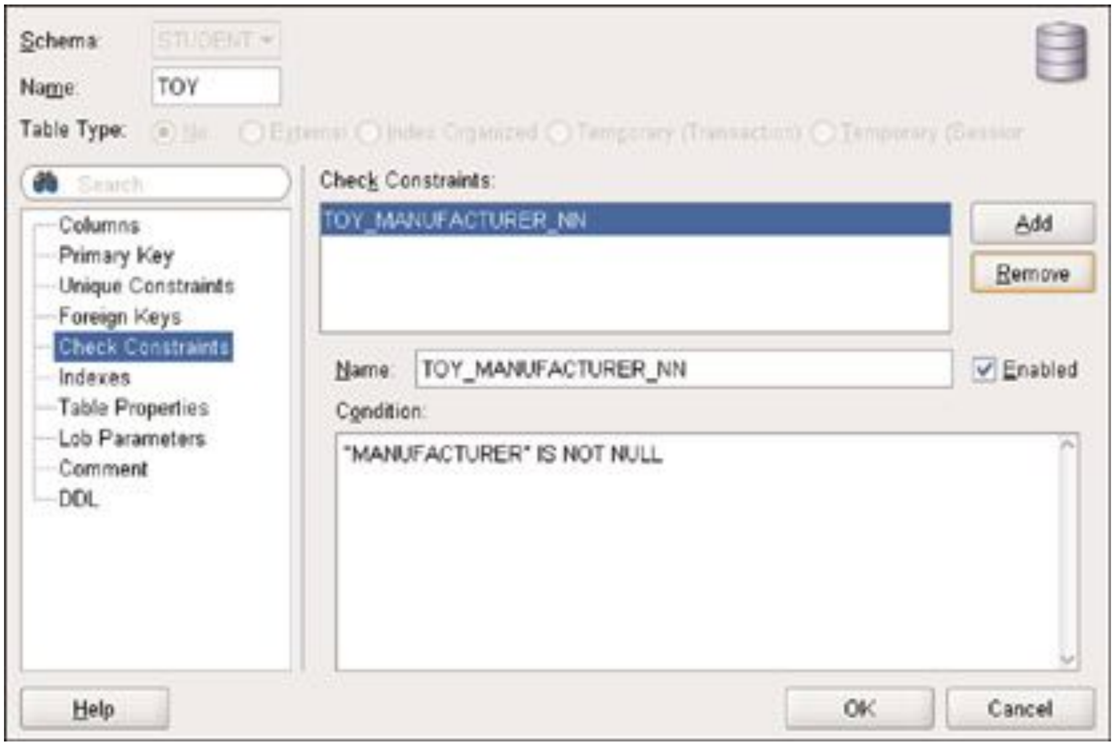
546

Using SQL Developer for ALTER TABLE Operations

SQL Developer allows you to perform many of the ALTER TABLE commands in this lab by using the menu options. Rather than requiring you to remember the syntax, the program generates the DDL. To modify any of the table or column characteristics, it is easiest to right-click on the table name in the Connection pane and to choose the Edit menu.

Using SQL Developer, you can add the constraint to the TOY table as shown in Figure 12.13. To obtain the appropriate ALTER TABLE statement, click on the DDL on the left side of the screen to obtain the generated DDL shown in Figure 12.14.

FIGURE 12.13 Adding a NOT NULL check constraint



TOY_MANUFACTURER_NN

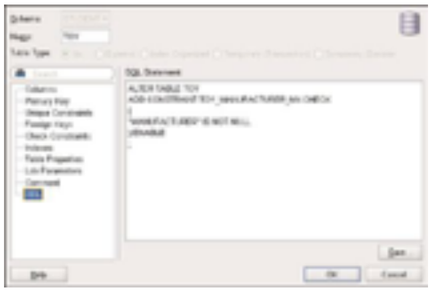
Name: TOY_MANUFACTURER_NN

☑ Enabled

Condition:

```
"MANUFACTURER" IS NOT NULL
```

FIGURE 12.14 DDL for adding a NOT NULL check constraint



```
ALTER TABLE TOY
ADD CONSTRAINT TOY_MANUFACTURER_NN CHECK
(
"MANUFACTURER" IS NOT NULL
)
ENABLE
;
```

You can only add a column together with a NOT NULL constraint if the table contains no data or if you specify a default. You will see examples of this in this lab's exercises.

Dropping Columns

You can drop columns from a table by using the ALTER TABLE command and the DROP clause. The following statement drops the LAST_PURCHASE_DATE column from the TOY table.

```
ALTER TABLE toy
   DROP (last_purchase_date)
Table altered.
```

If you want to drop multiple columns, separate the columns with commas.

```
ALTER TABLE toy
   DROP (manufacturer,
remaining_quantity)
Table altered.
```

Instead of dropping a column, you can mark it as unused with the SET UNUSED clause of the ALTER TABLE statement. With the current version of SQL Developer (version 1.5.3, as of this writing), the unused option is not available from a menu.

```
ALTER TABLE toy
   SET UNUSED (last_purchase_date)
Table altered.
```

Setting the column as unused is useful if you want to make the column no longer visible but do not want to physically remove it yet. When you issue a subsequent ALTER TABLE command with the DROP COLUMN clause or the ALTER TABLE command with the DROP UNUSED COLUMNS clause, Oracle physically removes the column from the database.

```
ALTER TABLE toy
   DROP UNUSED COLUMNS
Table altered.
```

Changing a column to UNUSED instead of dropping it is quicker, because it does not demand a lot of system resources. When the system is less busy, you can then physically remove the column.

Renaming Columns

You can rename an individual column with the following command.

```
ALTER TABLE toy RENAME COLUMN
description TO
   description_tx
```

Keep in mind that any dependent objects that reference this column become invalid.

547

548

Modifying Columns

You modify the data type, length, and column default of existing columns with the ALTER TABLE statement. There are a number of restrictions, as you'll see in the lab exercises.

The following statement changes the length of the DESCRIPTION column from 15 to 25 characters.

```
ALTER TABLE toy
   MODIFY (description
VARCHAR2(25))
Table altered.
```

The next statement modifies the data type of the REMAINING_QUANTITY column from NUMBER to VARCHAR2 and makes the column not null simultaneously. This statement executes successfully because the table contains no data.

```
ALTER TABLE toy
   MODIFY (remaining_quantity
VARCHAR2(6) NOT NULL)
Table altered.
```

You can also execute the statements individually, as in the following example.

```
ALTER TABLE toy
   MODIFY (remaining_quantity
VARCHAR2(6))
Table altered.

ALTER TABLE toy
   MODIFY (remaining_quantity NOT
NULL)
Table altered.
```

If you want to give the NOT NULL constraint a name, you use the following command. Instead of the system-generated SYS_ name, the constraint's name is then REMAIN_QT_NN.

```
ALTER TABLE toy
   MODIFY (remaining_quantity
   CONSTRAINT remain_qt_nn NOT
NULL)
Table altered.
```

Any changes to the structure of a table will invalidate other dependent database objects, such as views, triggers, and stored PL/SQL objects. The next time they are accessed, Oracle will attempt to compile or revalidate them. You can find the list of invalid objects in the ALL_OBJECTS or ALL_OBJECTS data dictionary view. Before making any changes, you can find out which objects are dependent on the table by querying the USER_DEPENDENCIES or ALL_ DEPENDENCIES data dictionary views or the SQL Developer Dependencies tab. Alternatively, you can use Extended Search to find out which objects reference the table. 548

549

Adding, Dropping, Disabling, and Enabling Constraints

Business rules are subject to occasional changes, and new constraints may be added or existing constraints modified. At times, you may temporarily disable a constraint for the purpose of loading data quickly and then re-enable the constraint afterward.

DROPPING CONSTRAINTS

When a constraint is no longer needed, you can drop it with the ALTER TABLE command and the DROP clause. The following statement drops a constraint by explicitly specifying the constraint name.

```
ALTER TABLE toy
   DROP CONSTRAINT toy_pk
Table altered.
```

Alternatively, you can drop a primary key constraint with the following statement.

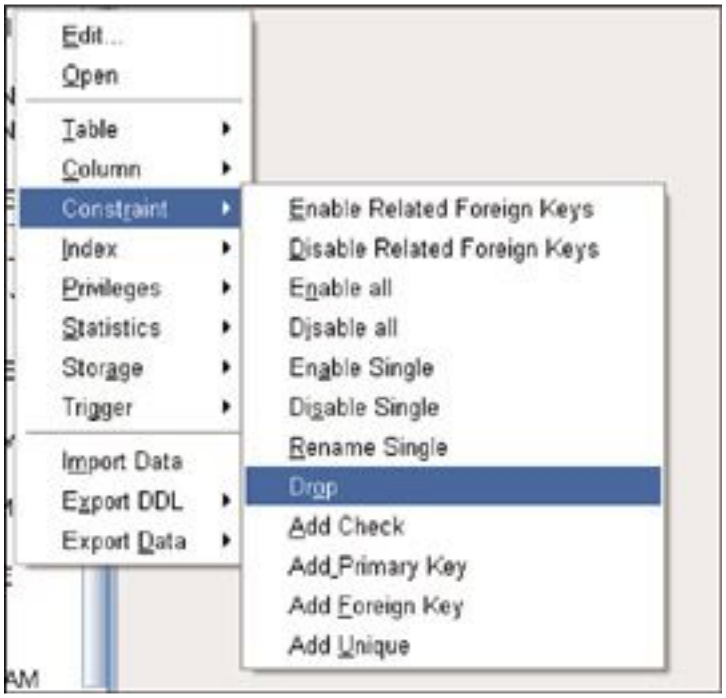
```
ALTER TABLE toy
   DROP PRIMARY KEY
```

If there is a unique constraint, you either issue the command with the constraint name or use a statement similar to the following.

```
ALTER TABLE toy
   DROP UNIQUE (description)
```

SQL Developer allows you to drop constraints by using the menu below the table node; choose Constraint, Drop (see Figure 12.15).

FIGURE 12.15 Dropping a constraint



ADDING CONSTRAINTS

You can add to a table any of the constraints you learned about in Lab 12.1 by using the ALTER TABLE…ADD command. When the TOY table was created, no primary key was specified. The following statement alters the TOY table to add a primary key constraint based on the TOY_ID column.

```
    ALTER TABLE toy
```

```
   ADD PRIMARY KEY(toy_id)
Table altered.
```

For unique and primary key constraints, Oracle checks for the existence of an index to enforce uniqueness of the constraint. If no index exists, Oracle automatically creates the index.

The preceding statement can be rewritten with a constraint name and a storage clause for the to-be-created index, as well as the tablespace on which the index is to be stored. This statement gives you control over the storage characteristics of the index.

```
ALTER TABLE toy
   ADD CONSTRAINT toy_pk PRIMARY
KEY(toy_id)
   USING INDEX TABLESPACE store_idx
   STORAGE (INITIAL 1M NEXT 500 K)
Table altered.
```

For performance reasons, you typically separate indexes and data by storing them on separate tablespaces, on different physical devices. The index is created in a tablespace called STORE_ IDX. Other characteristics of a table, such as its storage parameters and size, can also be specified with the ALTER TABLE command. In the previous example, 1MB of space is allocated, regardless of whether any rows exist. After this space is used, each subsequent amount of space allocated is 500K in size.

The following statement is an example of a SQL statement that creates the concatenated primary key constraint for the GRADE table, consisting of four columns. In addition, the space allocation and tablespace for the automatically associated unique index is located on the INDX table-space, and 100K is used for the initial extent.

```
ALTER TABLE grade
   ADD CONSTRAINT gr_pk PRIMARY KEY
   (student_id, section_id,
grade_type_code,
   grade_code_occurrence)
   USING INDEX TABLESPACE indx
   STORAGE (INITIAL 100K NEXT 100K)
```

ADDING A FOREIGN KEY

The following statement illustrates the creation of the two-column foreign key constraint on the GRADE table, referencing the concatenated primary key columns of the ENROLLMENT table.

```
ALTER TABLE grade
```

```
      ADD CONSTRAINT gr_enr_fk
   FOREIGN KEY
      (student_id, section_id)
      REFERENCES enrollment
   (student_id, section_id)
```

ADDING A SELF-REFERENCING FOREIGN KEY

The COURSE table has a recursive relationship; the PREREQUISITE column refers to the COURSE_NO column. It checks whether the values in the PREREQUISITE column are in fact valid COURSE_NO values. The following SQL statement shows the foreign key constraint command used to create the self-referencing constraint on the COURSE table.

```
   ALTER TABLE course
      ADD CONSTRAINT crse_crse_fk
   FOREIGN KEY (prerequisite)
      REFERENCES course (course_no)
```

If you are loading large amounts of data, you might want to consider temporarily disabling this constraint unless you can be sure that the sequence in which the data is inserted into the table is correct. The enabled constraint requires any courses that are prerequisites for other courses to be entered first.

ADDING A UNIQUE INDEX

The following example shows how the ALTER TABLE command on the SECTION table creates the unique index on the SECTION_NO and COURSE_NO columns. Because unique constraints automatically create an associated index, you want to place the index on a separate tablespace. Following is the syntax to place the index on the INDX tablespace, and the command also defines the initial and each subsequent extent.

```
ALTER TABLE section
   ADD CONSTRAINT sect_sect2_uk
   UNIQUE (section_no, course_no)
   USING INDEX TABLESPACE indx
   STORAGE
   (INITIAL 120K NEXT 120K)
```

ADDING A CHECK CONSTRAINT

The following statement adds a check constraint to the ZIPCODE table. It verifies that the entries in the ZIP primary key column are exactly five characters long and only numbers, not letters or special characters.

The TRANSLATE function converts each entered digit into a 9 and then checks whether the format equals 99999. Any nonnumeric digits are not translated; therefore, the result of TRANSLATE is unequal to 99999, and the value is rejected.

```
ALTER TABLE zipcode
   ADD CONSTRAINT zipcode_zip_ck
   CHECK (TRANSLATE(zip,
'1234567890', '9999999999') =
'99999')
```

Alternatively, you could come up with the following check constraint, but it has one drawback: A value such as '123.4' does not raise an error when the TO_NUMBER conversion function is applied. The LENGTH function is also fine because this is a string with a five-character length. There are many ways to handle data validity checking. You can also use a regular expression, which offers even more flexibility, as you will see shortly.

```
ALTER TABLE zipcode
   ADD CONSTRAINT zipcode_zip_ck
   CHECK (TO_NUMBER(zip)>0 AND
LENGTH(zip)= 5)
```

This check constraint is applied to the SALUTATION column of the INSTRUCTOR table.

```
ALTER TABLE instructor
   ADD CONSTRAINT
instructor_salutation_ck
   CHECK (salutation IN ('Dr',
'Hon', 'Mr', 'Ms', 'Rev') OR
salutation IS NULL)
```

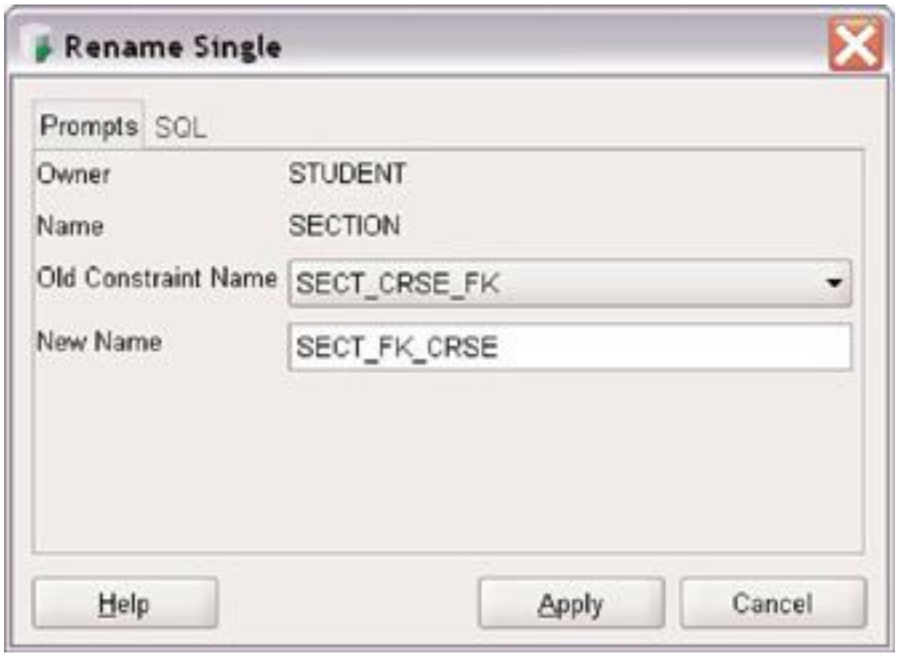
RENAMING CONSTRAINTS

You might want to use the RENAME CONSTRAINT command to change any system-generated constraint name to a more descriptive name. You can rename a constraint name with the following command.

```
ALTER TABLE section RENAME
CONSTRAINT sect_crse_fk TO
sect_fk_crse
```

In SQL Developer, you can access the Rename menu via the Actions… menu on SQL Developer's display of the table or from the Table node in the Connections pane. You see a Rename Single menu similar to Figure 12.6.

FIGURE 12.16 Renaming a constraint



Owner	STUDENT
Name	SECTION
Old Constraint Name	SECT_CRSE_FK
New Name	SECT_FK_CRSE

DISABLING AND ENABLING CONSTRAINTS

You can enable and disable constraints as necessary by using the ALTER TABLE command. By default, when a constraint is created, it is enabled, unless you explicitly disable it. You might want to disable constraints when updating massive volumes of data or inserting large amounts of data at once to decrease overall time for these operations. After the data manipulation is performed, you re-enable the constraint. 552

The following statement disables an existing primary key constraint named TOY_PK on the TOY table.

```
ALTER TABLE toy
   DISABLE CONSTRAINT toy_pk
```

Table altered.

When a primary key or unique constraint is disabled, by default any associated index is dropped. When the constraint is re-enabled, a unique index is re-created.

Alternatively, you can preserve the index of a unique or primary key if you specify the KEEP INDEX clause of the ALTER TABLE statement, as shown in the following statement.

```
ALTER TABLE toy
   DISABLE CONSTRAINT toy_pk KEEP
INDEX
```

When data changes are complete, you can enable the primary key with the following statement.

```
ALTER TABLE toy
   ENABLE PRIMARY KEY
```

Table altered.

Naming constraints helps when you want to disable or enable them. This statement explicitly specifies the constraint name and creates the index on a specified tablespace, with the listed storage parameters.

```
ALTER TABLE toy
   ENABLE CONSTRAINT toy_pk
   USING INDEX TABLESPACE store_idx
   STORAGE (INITIAL 1 M NEXT 500 K)
Table altered.
```

To find out the name of a constraint and its status (enabled or disabled), query the data dictionary views USER_CONSTRAINTS and USER_CONS_COLUMNS or use the SQL Developer Constraints tab.

If an index does not exist for a constraint, and you don't specify the tablespace name when you enable the constraint, the index is stored on your default tablespace, with the default storage size parameters. The storage clause on constraints is relevant only with primary and unique constraints because they create indexes.

The following statement disables the foreign key constraint between the COURSE and SECTION tables.

```
ALTER TABLE section
   DISABLE CONSTRAINT sect_crse_fk
Table altered.
```

If you want to disable multiple constraints, you can issue multiple statements or issue them in one ALTER TABLE statement. The individual DISABLE clauses are not separated by commas.

```
ALTER TABLE section
   DISABLE CONSTRAINT sect_crse_fk
   DISABLE CONSTRAINT sect_inst_fk
```

THE VALIDATE AND NOVALIDATE OPTIONS

As part of the ALTER TABLE statement to enable and disable constraints, Oracle also provides VALIDATE and NOVALIDATE options. The NOVALIDATE option enforces subsequent DML on the table complying with the constraint; existing data in the table can violate the constraint.

ENABLING/DISABLING ALL CONSTRAINTS AND RELATED FOREIGN KEYS

SQL Developer has menu options to disable or enable all constraints for a given table. This is useful when you load large volumes of data and you want to disable all the constraints. You do not need to disable/enable each constraint individually. The DDL tab reveals that SQL Developer generates a PL/SQL program to loop through the data dictionary to find all the constraints and then executes the commands to enable/disable each individual constraint.

You can also disable/enable all the related foreign keys with a separate menu option. If you execute this menu option for the ZIPCODE table, the foreign keys on the STUDENT and INSTRUCTOR tables related to the ZIP column are disabled.

Determining Which Rows Violate Constraints

Unless the NOVALIDATE option is used, when a constraint is re-enabled, Oracle checks to see if all the rows satisfy the condition of the constraint. If some rows violate the constraint, the statement fails, and Oracle issues an error message. The constraint cannot be enabled unless all exceptions are fixed or the offending rows are deleted.

FOREIGN KEY CONSTRAINT VIOLATIONS

If a row with a new course number was added to the SECTION table but the COURSE table has no such COURSE_NO, the foreign key constraint cannot be enabled, as indicated by the error message.

```
ALTER TABLE section
   ENABLE CONSTRAINT sect_crse_fk
ALTER TABLE section
*
ERROR at line 1:
ORA-02298: cannot validate
(STUDENT.SECT_CRSE_FK) - parent
keys not found
```

There are a variety of ways to determine which rows are the offending rows. For example, you can issue the following statement to display the rows.

```
SELECT course_no
   FROM section
```

```
MINUS
SELECT course_no
   FROM course
```

PRIMARY KEY CONSTRAINT VIOLATIONS

To determine which rows violate the primary key constraint, you can group by the primary key column and query for duplicates with the HAVING clause, as shown in the following SQL command.

```
SELECT section_id, COUNT(*)
   FROM section
GROUP BY section_id
HAVING COUNT(*) > 1
```

A subsequent DELETE operation of the duplicate rows might look as follows.

```
DELETE
   FROM section
   WHERE ROWID IN (SELECT
MAX(ROWID)
FROM section
GROUP BY section_id
HAVING COUNT(*) > 1)
```

The subquery identifies the duplicate SECTION_ID column values. The SELECT of the subquery retrieves the largest value of the ROWID pseudocolumn. Each Oracle table has a pseudocolumn called ROWID, which is not visible when describing the table or with a SELECT * statement. The ROWID is unique for every row, and this subquery statement picks the largest ROWID value, using the MAX function. The rows with these duplicate SECTION_ID values is deleted. (Make sure the non-primary key column values are identical, so you don't inadvertently delete rows that you want to keep.)

MORE WAYS TO IDENTIFY CONSTRAINT VIOLATIONS

Oracle allows you to record all the rows that violate a constraint in a table called EXCEPTIONS, and you can create the table with the Oracle script utlexcpt.sql, found in the %ORACLE_HOME%\rdbms\admin directory. You can then use the following syntax to place the violating rows in the EXCEPTIONS table.

```
ALTER TABLE tablename ENABLE
CONSTRAINT
```

```
     constraint_name EXCEPTIONS INTO
  exceptions
```

Writing Complex Check Constraints

You have seen a number of check constraint examples that validate data against simple logic. Sometimes a seemingly straightforward requirement can turn into a fairly complex check constraint. For example, imagine that you need to add validation to the PHONE column of 555 556 the STUDENT table to ensure that the entered number fits the ###-###-##### format. You can perform this validation with the TRANSLATE function (discussed in Lab 4.1), as you can see in the following statement.

```
ALTER TABLE student
   ADD CONSTRAINT student_phone_ck
CHECK
(TRANSLATE(phone,
'012345678',
'999999999') = '999-999-9999')
```

The TRANSLATE function determines whether any of the characters listed in the PHONE column have any of values of ‘0123456789’. When true, it translates them to the corresponding 9 value. Any character not listed is not translated and is retained as the original character. If the result of the TRANSLATE function equals to the pattern '999-999-9999', the value passes the check constraint.

If you want to include phone numbers without area codes or allow an optional period or space instead of the hyphen as a separation character, your check constraint quickly becomes complex, as you need to cover all the various combinations with OR conditions. You may even end up writing a trigger to make the logic more transparent.

Oracle includes regular expressions capabilities in the SQL language. A *regular expression* is pattern-matching functionality found in many programming languages. While regular expressions look fairly complex initially, you will appreciate their power and flexibility when you understand the meaning of the metacharacters that make up the regular expression. The following example duplicates the identical functionality of the TRANSLATE function to illustrate the meaning of some of the metacharacters.

```
ALTER TABLE student
   ADD CONSTRAINT student_phone_ck
CHECK
 (REGEXP_LIKE(phone,
 '^[[:digit:]]{3}-[[:digit:]]{3}-
 [[:digit:]]{4}$'))
```

The REGEXP_LIKE operator is similar to the LIKE operator in that it checks whether a pattern is found. The first metacharacter, ^, indicates that there may not be any characters before the pattern, just like the $ at the end of the regular expression indicates that there may not be any extra characters at the end of the line. The [[:digit:]] character class and list specify that only a digit is allowed. The digit class is repeated three times, as indicated with the {3} repetition operator. This part of the regular expression represents the area code. The pattern continues with a hyphen, followed by another three digits, a hyphen, and then another four digits.

If characters other than a hyphen are valid in your phone number format, you can modify the regular expression by including those characters inside a character list enclosed by square brackets, [ ]. In this instance, spaces, periods, and dashes are allowed, or consecutive numbers without any separator. The character list now reads as [-.]?, with the? metacharacter indicating that 0 or 1 repetition of this class list is allowed.

```
ALTER TABLE student
```

```
    ADD CONSTRAINT student_phone_ck
CHECK
(REGEXP_LIKE(phone,
'^[[:digit:]]{3}[-. ]?[[:digit:]]
{3}[-. ]?[[:digit:]]{4}$'))
```

You can take the regular expression functionality a step further and add validation for phone extension numbers, area codes beginning only with numbers 2 through 9, optional area code parentheses, and alphanumeric formats such as 800-REGEXPR. In Chapter 16, "Regular Expressions and Hierarchical Queries," you will learn more about the many other metacharacters that help you validate patterns such as zip codes, e-mail addresses, or URLs.

Restrictions on Check Constraints

Check constraints impose a few restrictions you should be aware of. A check constraint cannot refer to a column in another table; only columns within the same table are allowed. You cannot use subqueries or scalar subquery expressions. References to nondeterministic functions such as SYSDATE, USER, and CURRENT_TIMESTAMP are not allowed because they may return different results each time they are called. (You can use them for default values.) The pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM are also not permitted.

You can overcome these restrictions by writing triggers using PL/SQL. For example, you can write a trigger to reference columns in other tables to validate data, call the SYSDATE function to fill in the current date and time into the LASTMOD_DT column, or automatically create primary key column values that reference the NEXTVAL pseudocolumn of a sequence. At the beginning of this chapter, a trigger example illustrates the use of the SYSDATE function; you will see another PL/SQL trigger example referencing a sequence in Chapter 13.

Read-only Tables

For any table you own, you can modify the data or alter the object. Oracle 11*g* introduced the ability to make a table read-only; this prevents even the owner from performing any data manipulations or issuing any changes to the structure of the table. To make a table read-only or return it to write mode, you use the following syntax options.

```
ALTER TABLE tablename READ ONLY
ALTER TABLE tablename READ WRITE
```

You can still drop the table from the schema. If you accidentally drop a table, and you might be able to restore it from the recycle bin.

LAB 12.2 EXERCISES

a) Alter the table called NEW_STUDENT that you created in exercise b in Lab 12.1 by adding four columns. The columns should be called PHONE, NUM_COURSES with data type and length NUMBER(3), CREATED_BY, and CREATED_DATE. Determine the other column data types and lengths, based on the STUDENT table. The PHONE, NUM_COURSES, and CREATED_BY columns should allow null values, with the CREATED_BY column defaulting to the user's login name. The CREATED_DATE column should not allow null values and should default to today's date. Describe the table when you have finished.

b) Execute the following INSERT statement to insert a row into the NEW_STUDENT table.

Then alter the table to change the PHONE column from NULL to NOT NULL. What do you observe?

```
INSERT INTO new_student
   (first_name, last_name,
description, registration_date)
VALUES
   ('Joe', 'Fisher', 'Intro to
Linux', SYSDATE)
```

c) Alter the NEW_STUDENT table to change the REGISTRATION_DATE column from the DATE data type to the VARCHAR2 data type. What do you observe?

d) Alter the NEW_STUDENT table to create a primary key consisting of the FIRST_NAME and LAST_NAME columns.

e) Alter the NEW_STUDENT table to change the length of the LAST_NAME column from 25 to 2. What do you observe?

f) Disable the primary key constraint on the NEW_STUDENT table and write an INSERT statement with the value Joe Fisher for the first and last names to prove it is successful. Then enable the constraint again and describe the result.

g) Add to the NEW_STUDENT table the column STUDY_DURATION of data type INTERVAL YEAR TO MONTH and the column ALUMNI_JOIN_DATE with the data type TIMESTAMP WITH TIME ZONE and a six-digit precision.

h) Drop the foreign key constraint FORMER_STUDENT_ZIPCODE_FK on the FORMER_STUDENT table and change it to an ON DELETE SET NULL foreign key constraint. Test the behavior by inserting a new zip code in the ZIPCODE table and creating a new student row with this new zip code and then deleting the same zip code from the ZIPCODE table. Query the FORMER_STUDENT table to see the effect.

i) Drop all the tables created throughout the labs. The table names are STUDENT_CANDIDATE, NEW_STUDENT, COURSE2, EXTINCT_ANIMAL, and FORMER_STUDENT.

LAB 12.2 EXERCISE ANSWERS

a) Alter the table called NEW_STUDENT that you created in exercise b in Lab 12.1 by adding four columns. The columns should be called PHONE, NUM_COURSES with data type and length NUMBER(3), CREATED_BY, and CREATED_DATE. Determine the other column data types and lengths, based on the STUDENT table. The PHONE, NUM_COURSES, and CREATED_BY columns should allow null values, with the CREATED_BY column defaulting to the user's login name. The CREATED_DATE column should not allow null values and should default to today's date. Describe the table when you have finished. 558

559

ANSWER: You add the four columns by using a single ALTER TABLE command, separating the column names with commas. The CREATED_BY column has a DEFAULT clause to default the column to the value of the user's login name; the CREATED_DATE column contains a NOT NULL constraint and defaults the column to the value SYSDATE.

```
ALTER TABLE new_student
  ADD (phone VARCHAR2(15),
       num_courses NUMBER(3),
       created_by VARCHAR2(30) DEFAULT USER,
       created_date DATE DEFAULT SYSDATE NOT NULL)
Table altered.

SQL> DESC new_student
 Name                              Null?    Type
 --------------------------------- -------- ------------
 FIRST_NAME                                 VARCHAR2(25)
 LAST_NAME                                  VARCHAR2(25)
 DESCRIPTION                                VARCHAR2(50)
 REGISTRATION_DATE                          DATE
 PHONE                                      VARCHAR2(15)
 NUM_COURSES                                NUMBER(3)
 CREATED_BY                                 VARCHAR2(30)
 CREATED_DATE                      NOT NULL DATE
```

A column or columns can be added to a table, regardless of whether the table contains data. However, you cannot add columns with a NOT NULL constraint if the column contains NULL values. Therefore, you must first add the column with the NULL constraint, update the column with values, and then alter the table to modify the column to add the NOT NULL constraint. Starting with Oracle 11*g*, you can make use of the DEFAULT clause to avoid having to update the table, as you'll see shortly.

SETTING COLUMNS TO THE DEFAULT VALUES

How do default values behave when you insert data into a column? The CREATED_DATE column has a default value of SYSDATE. If you want this default value to appear in an INSERT statement, you can either not list the column in the INSERT statement or you can explicitly state the DEFAULT keyword.

In the next example, the CREATED_DATE column is explicitly specified, and the DEFAULT keyword is used to place the current date and time into the column. The CREATED_BY default returns the value of the USER function, which is the name of the user currently logged in. Because it's not listed in the following INSERT statement, this default value is used.

```
INSERT INTO new_student
   (first_name, last_name,
description, created_date)
VALUES
   ('Julian', 'Soehner', 'Test#1',
DEFAULT)
SELECT description, created_by,
created_date
   FROM new_student
```

```
DESCRIPTION     CREATED_BY                        CREATED_D
--------------- ------------------------------ ---------
Test#1          STUDENT                           18-MAY-09

1 row selected.

ROLLBACK
Rollback complete.
```

MODIFYING OR REMOVING COLUMN DEFAULT VALUES

A column with a DEFAULT option can be removed or changed to another default value. The following example removes the default value for the CREATED_BY column and changes the value for the CREATED_DATE column to '01-JAN-2009'.

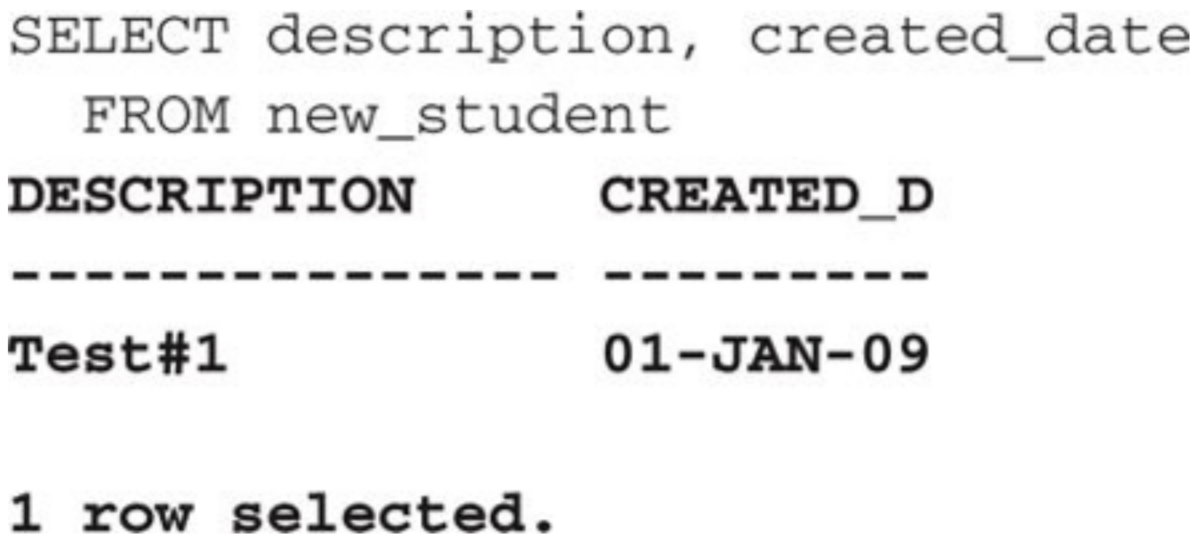
```
ALTER TABLE new_student
   MODIFY (created_by VARCHAR2(30)
DEFAULT NULL, created_date DATE
DEFAULT TO_DATE('01-Jan-2009'))
Table altered.
```

A DDL command such as this ALTER TABLE command cannot be rolled back, and it issues an implicit COMMIT.

The following example shows an update of a column to reset its present value to the default value using the DEFAULT keyword.

```
UPDATE new_student
   SET created_date = DEFAULT
WHERE description = ‘Test#1’
```

The result now shows the CREATED_DATE column with the value 01-Jan-2009.



```
SELECT description, created_date
  FROM new_student
DESCRIPTION      CREATED_D
---------------- ---------
Test#1           01-JAN-09


1 row selected.
```

New with Oracle 11*g*, you do not have to update the column first want to add a NOT NULL constraint to a table with existing data. The default value is stored in the data dictionary, and when the user selects the column, the default value is retrieved from the data dictionary.

This new feature is particularly helpful if the table has many rows; it avoids time-consuming updates and the associated the generation of large undo segments and redo logs.

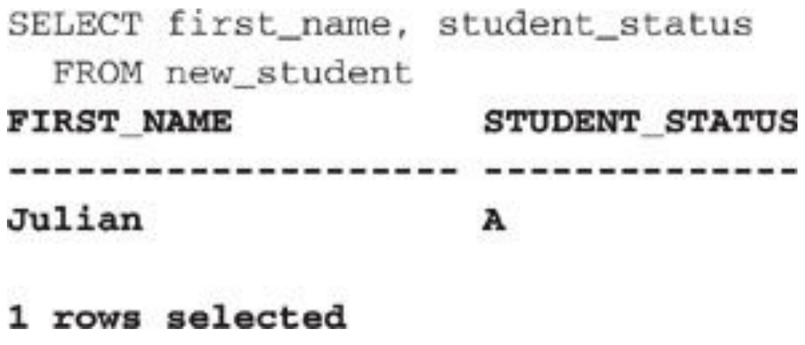
Following is an example of adding a column along with the new associated column default performed against an Oracle 11*g* database.

560

561

```
ALTER TABLE new_student
   ADD (student_status VARCHAR2(2)
DEFAULT ‘A′ NOT NULL)
```

A subsequent query against the table’s newly added STUDENT_STATUS column displays the column default.



```
SELECT first_name, student_status
  FROM new_student
FIRST_NAME           STUDENT_STATUS
-------------------- --------------
Julian               A

1 rows selected
```

b) Execute the following INSERT statement to insert a row in the NEW_STUDENT table. Then alter the table to change the PHONE column from NULL to NOT NULL. What do you observe?

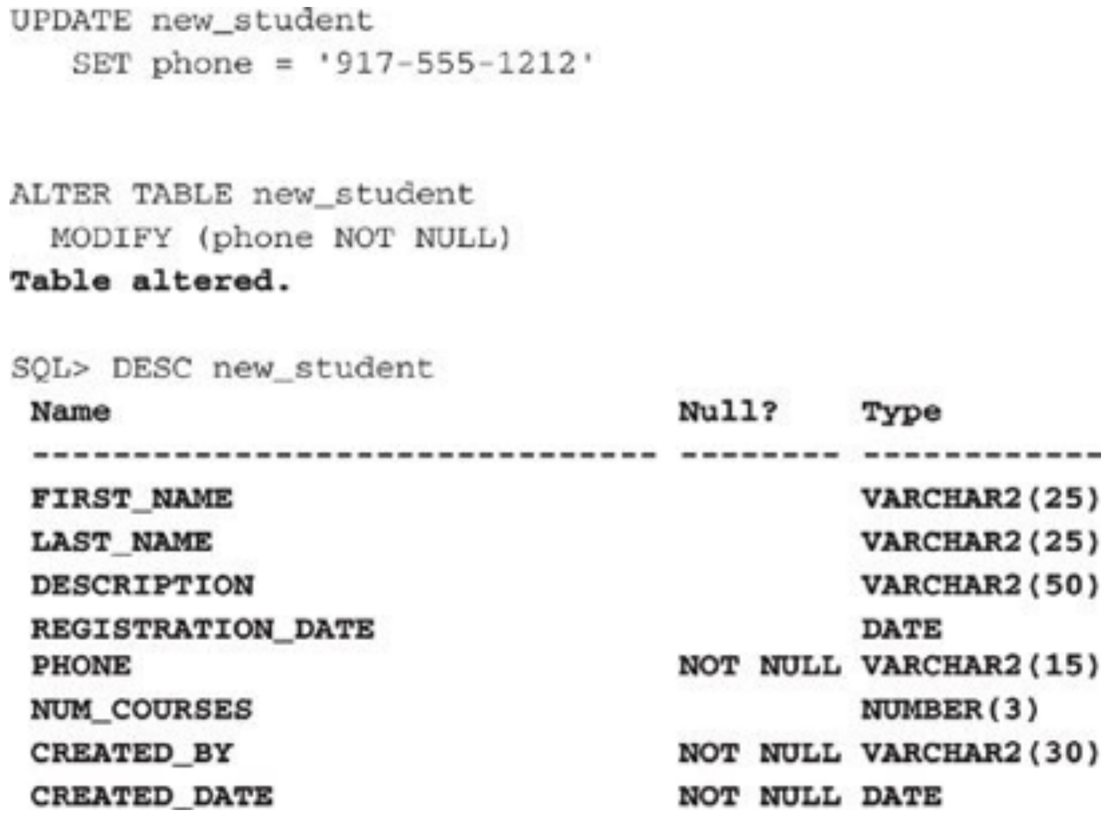
```
INSERT INTO new_student
   (first_name, last_name,
description, registration_date)
VALUES
   ('Joe', 'Fisher', 'Intro to
Linux', SYSDATE)
```

ANSWER: The column cannot be modified to have a NOT NULL constraint because there is already a row in the table that contains a NULL value in the column.

```
ALTER TABLE new_student
   MODIFY (phone NOT NULL)
MODIFY (phone NOT NULL)
*
ERROR at line 2:
ORA-02296: cannot enable
(STUDENT.) - null values found
```

You cannot modify an existing column to NOT NULL if it contains NULL values. You must first add data to the column and then modify the column to add the constraint. If you use Oracle

11*g*,you can add a default value without having to update all the rows (see exercise a).



```
UPDATE new_student
   SET phone = '917-555-1212'


ALTER TABLE new_student
  MODIFY (phone NOT NULL)
Table altered.

SQL> DESC new_student
 Name                          Null?    Type
 ----------------------------- -------- -----------
 FIRST_NAME                             VARCHAR2(25)
 LAST_NAME                              VARCHAR2(25)
 DESCRIPTION                            VARCHAR2(50)
 REGISTRATION_DATE                      DATE
 PHONE                         NOT NULL VARCHAR2(15)
 NUM_COURSES                            NUMBER(3)
 CREATED_BY                    NOT NULL VARCHAR2(30)
 CREATED_DATE                  NOT NULL DATE
```

```
PHONE NOT NULL
VARCHAR2(15)NUM_COURSES
NUMBER(3)CREATED_BY NOT NULL
VARCHAR2(30)CREATED_DATE NOT NULL
DATE
```

You can also change the column back to NULL with the following statement.

```
ALTER TABLE new_student
   MODIFY (phone NULL)
Table altered.
```

```
ALTER TABLE new_student
  MODIFY (phone NULL)
Table altered.

SQL> DESC new_student
 Name                              Null?    Type
 --------------------------------- -------- -----------
 FIRST_NAME                                 VARCHAR2(25)
 LAST_NAME                                  VARCHAR2(25)
 DESCRIPTION                                VARCHAR2(50)
 REGISTRATION_DATE                          DATE
 PHONE                                      VARCHAR2(15)
 NUM_COURSES                                NUMBER(3)
 CREATED_BY                        NOT NULL VARCHAR2(30)
 CREATED_DATE                      NOT NULL DATE
```

DEFINING A COLUMN AS NULL VERSUS NOT NULL

Deciding whether a column should be NOT NULL or NULL leads into a discussion of nulls in general. A column that allows null values is subject to different interpretations. If you find a null value in a column, it might mean many things: Perhaps the value is simply unknown, unspecified (the user didn't choose any of the available choices), or perhaps not applicable. An encoded value can help distinguish between these differences.

Suppose you have a table that holds client demographic data including a GENDER column. There simply aren't just two genders—male and female. What if your client is not an individual, but a corporation? Do you enter a null value, and does a null value mean not applicable? What if the gender is unknown? You can come up with a lot of other scenarios for a seemingly simple GENDER column. Therefore, database designers use consistent values throughout to ensure that null values are interpreted correctly. For example, you can enter the value? for unknown, N/A for not applicable, and OTH for other, or you can create a default value for unspecified.

Writing queries against data that contains null values poses another challenge. Unless you specifically use the IS NULL operator (or the NVL or COALESCE function) on a column, null values are ignored. You must always keep in mind the possibility of null values when dealing with data. Furthermore, if you apply a function to an indexed column or query using the IS NULL or IS NOT NULL operator, the query won't be able to take advantage of the index. Nulls can also have positive effects. An example of this is an order status flag column on an order table, indicating whether the order needs to be processed. If you enter YES, it indicates that the order is incomplete; if you enter NO, it indicates that the order is processed. If instead you allow only YES and a null value, you can actually improve the performances of queries looking for orders to be processed, because you can build an index on this status flag column, and only non-null entries are stored in the index, which is rather small because there are few entries. Values are then retrieved quickly. 562

563

c) Alter the NEW_STUDENT table to change the REGISTRATION_DATE column from the DATE data type to the VARCHAR2 data type. What do you observe?

ANSWER: A column's data type cannot be changed when there is data in the column.

```
ALTER TABLE new_student
MODIFY (registration_date
VARCHAR2(12))
MODIFY (registration_date
VARCHAR2(12))
*ERROR at line 2:ORA-01439: column
to be modified must be empty to
change datatype
```

CHANGING A COLUMN'S DATA TYPE

It is possible to change a column's data type in two circumstances. The first circumstance is when changing from one data type to a compatible data type, such as

VARCHAR2 to CHAR. The following statement changes the REGISTRATION_DATE column from the DATE data type to the compatible TIMESTAMP data type. You can't change from a TIMESTAMP back to a DATE data type unless the column is null.

```
ALTER TABLE new_student
   MODIFY (registration_date
TIMESTAMP(3))
Table altered.
```

The second circumstance is when the column is empty, as in the following example. This statement sets the column to NULL to facilitate the change to a completely different data type.

```
UPDATE new_student
   SET registration_date = NULL
1 row updated.
```

```
ALTER TABLE new_student
   MODIFY (registration_date
VARCHAR2(12))
Table altered.
```

d) Alter the NEW_STUDENT table to create a primary key consisting of the FIRST_NAME and LAST_NAME columns.

ANSWER: The NEW_STUDENT table is altered to add a PRIMARY KEY constraint consisting of the two columns.

```
ALTER TABLE new_student
   ADD CONSTRAINT new_student_pk
PRIMARY KEY(first_name, last_name)
Table altered.
```

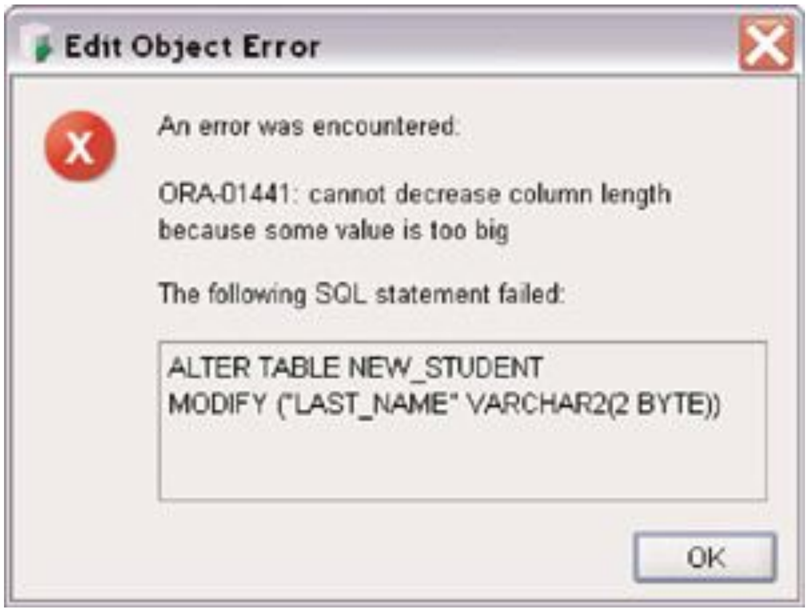
The ADD PRIMARY KEY keywords are used to add the primary key constraint.

Actually, the choice of this primary key is not a very good one, aside from students having the same name, because a name entered in all uppercase is considered different from a name entered in mixed case.

e) Alter the NEW_STUDENT table to change the length of the LAST_NAME column from 25 to 2. What do you observe?

ANSWER: The length of a column cannot be decreased when the existing values in the column are larger than the new column width. Figure 12.17 shows the error message and the failed SQL statement.

FIGURE 12.17 Error message after an attempt to decrease column length



Edit Object Error

An error was encountered:

ORA-01441: cannot decrease column length because some value is too big

The following SQL statement failed:

```
ALTER TABLE NEW_STUDENT
MODIFY ("LAST_NAME" VARCHAR2(2 BYTE))
```

OK

For columns containing data, the length of the column can always be increased, as in the following example, but it cannot be decreased if existing data is larger than the new column width.

```
ALTER TABLE new_student
   MODIFY (last_name VARCHAR2(30))
```
`Table altered.`

f) Disable the primary key constraint on the NEW_STUDENT table and write an INSERT statement with the value Joe Fisher for the first and last names to prove it is successful. Then enable the constraint again and describe the result.

ANSWER: The value Joe Fisher exists twice in the FIRST_NAME and LAST_NAME columns so the primary key constraint cannot be enabled on the table.

```
ALTER TABLE new_student
   DISABLE PRIMARY KEY
```
```
Table altered.
```

```
INSERT INTO new_student
   (first_name, last_name, phone,
created_by, created_date)
VALUES
   ('Joe', 'Fisher',
'718-555-1212', USER, SYSDATE)
```
```
1 row created.
```

```
ALTER TABLE new_student
ENABLE PRIMARY KEY
```
```
ALTER TABLE new_student*ERROR at
line 1:
ORA-02437: cannot enable
(STUDENT.NEW_STUDENT_PK) - primary
key violated
```

It is dangerous to disable a table’s primary key because the integrity of the data may be violated. The only time you might want to temporarily disable constraints is when you are performing large data loads. Otherwise, if the constraints are enabled, each row must be evaluated to ensure that it does not violate any of the constraints, thus slowing down the data-loading process. After the data load or update is done, you re-enable the constraints.

Dropping all constraints may not always help speed-up all data updates, particularly if some constraints are associated with indexes that are used as part of an UPDATE’s WHERE clause.

g) Add to the NEW_STUDENT table the column STUDY_DURATION of data type INTERVAL YEAR TO MONTH and the column ALUMNI_JOIN_DATE with the data type TIMESTAMP WITH TIME ZONE and a six-digit precision.

ANSWER: The ALTER TABLE statement adds both columns simultaneously. The six-digit fractional seconds are the default for the TIMESTAMP WITH TIME ZONE data type and do not need to be specified explicitly.

```
ALTER TABLE new_student
   ADD (study_duration INTERVAL
YEAR TO MONTH, alumni_join_date
TIMESTAMP (6) WITH TIME ZONE)
Table altered.
```

h) Drop the foreign key constraint FORMER_STUDENT_ZIPCODE_FK on the FORMER_STUDENT table and then change it to an ON DELETE SET NULL foreign key constraint. Test the behavior by inserting a new zip code in the ZIPCODE table and creating a new student row with this new zip code and then deleting the same zip code from the ZIPCODE table. Query the FORMER_STUDENT table to see the effect.

ANSWER: The DROP CONSTRAINT clause removes the constraint, and you can then add the foreign key with the ON DELETE SET NULL constraint instead.

```
ALTER TABLE former_student
   DROP CONSTRAINT
former_student_zipcode_fk
```

Table altered.

```
ALTER TABLE former_student
   ADD CONSTRAINT
former_student_zipcode_fk
FOREIGN KEY(zip)
REFERENCES zipcode (ZIP) ON DELETE
SET NULL
```

Table altered.

The follow example inserts a new zip code with the value 90210 into the ZIPCODE table.

```
INSERT INTO zipcode
(zip, city, state, created_by,
created_date, modified_by,
modified_date)
VALUES
('90210','Hollywood', 'CA',
'Alice',
sysdate, 'Alice', sysdate);
```

1 row created.

565

566

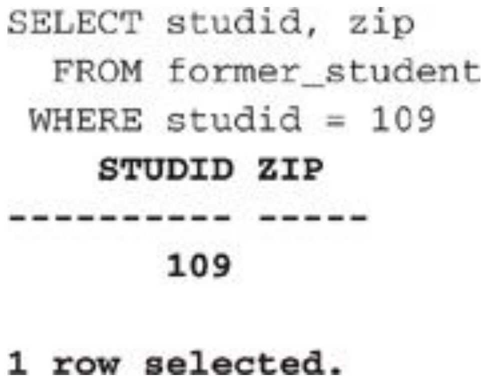
To demonstrate the functionality, insert a zip code into the FORMER_STUDENT table.

```
INSERT INTO former_student
   (studid, first_nm, last_nm,
enrolled, zip)
VALUES
   (109, ‘Alice’, ‘Rischert’, 3,
‘90210’)
1 row created.
```

Now delete zip code 90210 from the ZIPCODE table.

```
DELETE FROM zipcode
   WHERE zip = ‘90210’
1 row deleted.
```

A query against the FORMER_STUDENT table reveals the effect; the column ZIP is updated to a null value. The ZIP column must permit entry of null values.



```
SELECT studid, zip
  FROM former_student
 WHERE studid = 109
    STUDID ZIP
---------- -----
       109

1 row selected.
```

If you attempt to delete a row that exists not just in the NEW_STUDENT table but perhaps also in the

STUDENT or INSTRUCTOR table, such as the value 10025, you will be prevented from using the DELETE operation because these other tables are referencing ZIPCODE with a DELETE restrict. In this case, the INSTRUCTOR table is referencing this value as well, and orphan rows are not allowed.

```
DELETE FROM zipcode
   WHERE zip = '10025'
DELETE FROM zipcode*ERROR at line
1:
ORA-02292: integrity constraint
(STUDENT.INST_ZIP_FK)
violated - child record found
```

If your schema name is not STUDENT, but a different account name, the constraint name error will be prefixed with the respective schema/user name.

The other foreign key alternatives to the ON DELETE SET NULL options are the two statements listed next. The first adds the DELETE

RESTRICT default, and the second shows the ON DELETE CASCADE constraint alternative.

```
ALTER TABLE former_student
   ADD CONSTRAINT
former_student_zipcode_fk
FOREIGN KEY(zip)
REFERENCES zipcode (ZIP)
```

```
ALTER TABLE former_student
   ADD CONSTRAINT
former_student_zipcode_fk
FOREIGN KEY(zip)
REFERENCES zipcode (ZIP) ON DELETE
CASCADE
```

The foreign key constraint enforces the relationship between the tables also with respect to insertions and updates. Only values found in the parent table are allowed. Null values are allowed if the foreign key column is defined as NULL.

i) Drop all the tables created throughout the labs. The table names are STUDENT_CANDIDATE, NEW_STUDENT, COURSE2, EXTINCT_ANIMAL, and FORMER_STUDENT.

ANSWER: Use the DROP TABLE command to remove the tables from the schema.

```
DROP TABLE student_candidate
```
Table dropped.

```
DROP TABLE new_student
```
Table dropped.

```
DROP TABLE course2
```
Table dropped.

```
DROP TABLE extinct_animal
```
Table dropped.

```
DROP TABLE former_student
```
Table dropped.

Lab 12.2 Quiz

In order to test your progress, you should be able to answer the following questions.

1) The following ALTER TABLE statement contains an error.

```
ALTER TABLE new_student
   DROP CONSTRAINT PRIMARY_KEY
```

_______**a)** True

_______**b)** False

2) The ADD and MODIFY keywords can be used interchangeably in an ALTER TABLE statement.

_______**a)** True

_______**b)** False

3) You can always add a NOT NULL constraint.

_______**a)** True

_______**b)** False

4) A constraint must have a name in order for it to be disabled.

_______**a)** True

_______**b)** False

5) A column's data type can be changed only when the column contains no data.

_______**a)** True

_______**b)** False

6) On a read-only table, no DDL commands except for DROP TABLE and ALTER TABLE to READ WRITE are permitted.

_______**a)** True

_______**b)** False

ANSWERS APPEAR IN APPENDIX A.

WORKSHOP

The projects in this section are meant to prompt you to utilize all the skills you have acquired throughout this chapter. The answers to these projects can be found at the companion Web site to this book, located at **www.oraclesqlbyexample.com**.

1) Create a table called TEMP_STUDENT with the following columns and constraints: a column STUDID for student ID that is NOT NULL and is the primary key, a column FIRST_NAME for student first name, a column LAST_NAME for student last name, a column ZIP that is a foreign key to the ZIP column in the ZIPCODE table, and a column REGISTRATION_DATE that is NOT NULL and has a CHECK constraint to restrict the registration date to dates after January 1, 2000.

2) Write an INSERT statement that violates one of the constraints for the TEMP_STUDENT table you created in exercise 1. Write another INSERT statement that succeeds when executed and commit your work.

3) Alter the TEMP_STUDENT table to add two more columns called EMPLOYER and EMPLOYER_ZIP. The EMPLOYER_ZIP column should have a foreign key constraint that references the ZIP column of the ZIPCODE table. Update the EMPLOYER column and alter the table once again to make the EMPLOYER column NOT NULL. Drop the TEMP_STUDENT table when you are done with the exercise.