The University of Texas at Austin
McCombs School of Business

MIS 381N INTRO. TO DATABASE MANAGEMENT

PL/SQL

Tayfun Keskin
Visiting Clinical Professor, The University of Texas at Austin, McCombs School of Business
Associate Teaching Professor, University of Washington Seattle, Foster School of Business

QUESTIONS

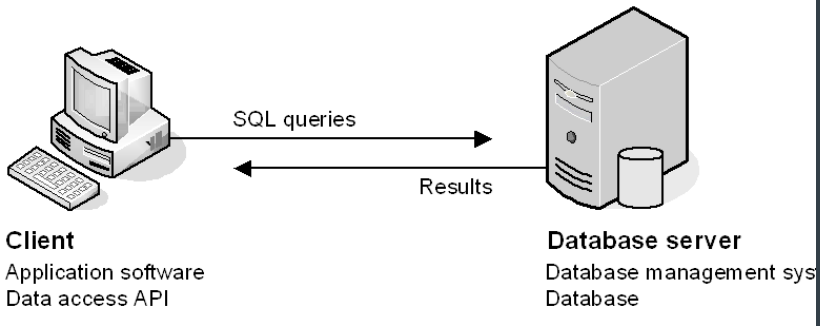
- Any questions before we begin?

AGENDA

Lecture	PL/SQL Data Governance
Hands-On	Exercises
Looking Forward	Homework 4 Exam 2

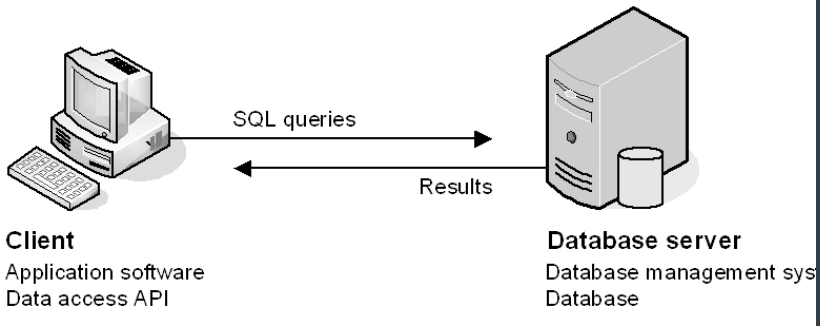
REMINDER QUESTION

Where do you write and run SQL?



REMINDER QUESTION

Where do tables get created and saved?



Oracle 19c has a Programming Language Built in

- It's called PL/SQL
- It stands for **"Procedural Language extensions to the Structured Query Language"**
- Expands your ability to fine tune control of the Oracle database
- PL/SQL is a full 3rd generation programming language, such as C++

WHAT CAN I DO WITH PL/SQL?

- Declare variables
- Limiting the scope of code blocks
- Using loops and conditional statements (`FOR, WHILE, IF`)
- Call external functions
- Processing results

WHY IS PL/SQL IMPORTANT?

- The integration of Oracle DB and PL/SQL is very tight
- All data types used by one are available to the other
  - Such as VARCHAR2, NUMBER, DATE
- Transparently uses declared data types with %TYPE and %ROWTYPE without knowing the exact data type at runtime

QUESTION

What is the difference between a procedural and declarative programming language?

(Hint: SQL is primarily declarative)

DIFFERENCE BETWEEN SQL AND PL/SQL

- SQL is declarative:
  - You write the code as “Here’s what I need to do”
  - Optimizer figures out the best way
- PL/SQL is procedural:
  - You write the code as “Here’s how to do what I want”
  - Optimizer is not involved

PL/SQL FEATURES

- Functions: process zero or more variables and return a variable of any data type
- Procedures: process zero or more arguments in a stored block of code
- Variable declarations: common data types and cursors
- FOR and WHILE loops with CONTINUE and EXIT
- IF-THEN-ELSE statement
- Exception handling

WHAT IS A DB FUNCTION?

- A sequence of SQL and PL/SQL statements stored by name
- Stored in the DB’s data dictionary that you can call again later
- It can have zero arguments or dozens (usually a few)
- Returns one value of any datatype, even a pointer
- A.K.A.: user function, user-defined function, stored function

CREATE FUNCTION SYNTAX

```
CREATE [OR REPLACE] FUNCTION function_name
[(
    parameter_name_1 data_type
    [, parameter_name_2 data_type]...
)]
RETURN data_type
{IS | AS}
pl_sql_block
```

```
-- Example: A function that returns a vendor ID
CREATE OR REPLACE FUNCTION get_vendor_id
(
   vendor_name_param VARCHAR2
)
RETURN NUMBER
AS
  vendor_id_var NUMBER;
BEGIN
  SELECT vendor_id
  INTO vendor_id_var
  FROM vendors
  WHERE vendor_name = vendor_name_param;

  RETURN vendor_id_var;
END;
/
```

QUESTION

If your function compiles without any errors, does it mean your function is without an error?

Or, useful?

WHAT IS A DB PROCEDURE?

- Companion to stored functions
- A sequence of SQL and PL/SQL statements stored by name
- It can be anonymous (it doesn't get stored permanently)
- It can have zero or more arguments, but **do not return values**
- It can be stored by itself, or stored in a package

WHAT? NO RETURN VALUE?

- It cannot be called directly from a SELECT statement (because it’s not a function, it’s not part of the SQL language)
- You can still run it with an EXEC or CALL statement
- You can modify and return values if one of the parameters is declared as OUT in the argument list (not very common)

CREATE PROCEDURE SYNTAX

```
CREATE [OR REPLACE] PROCEDURE proc_name
[(
    parameter_name_1 data_type
    [, parameter_name_2 data_type]...
)]

{IS | AS}

pl_sql_block
```

```
-- Example: A sproc that updates a table
CREATE OR REPLACE PROCEDURE
update_invoices_credit_total
(
  invoice_number_param  VARCHAR2,
  credit_total_param    NUMBER
)
AS
BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```

TRUE OR FALSE?

A stored procedure can modify and return parameters defined as **OUT**

WHAT IS A VARIABLE?

- A variable is a meaningful name which facilitates a programmer to store data temporarily during the execution of code
- Declaring variables is optional, but you'll need to put something in the declaration section to take full advantage of PL/SQL
- A variable can be CONSTANT (I know it's ironic)

DATA TYPES ALLOWED

- All data types are allowed in the declaration section
- If you don't want to change the code, you can define a variable with the `%TYPE` keyword (it automatically finds the data type of the column at runtime)
- You can define an entire row/table with `%ROWTYPE`

DECLARING VARIABLES

- A procedure of any type can have an optional DECLARE section
- You can only DECLARE in an anonymous block; otherwise, use IS
- Declared variables can be initialized, uninitialized (NULL)
  - A constant, by definition, is always initialized
- If you want to avoid changing your code as your table or column attributes change, use %TYPE or %ROWTYPE keywords and it will help you avoid recompilations in the future

WHAT IS A CURSOR?

- When an SQL statement is processed, Oracle creates a memory area known as context area
- A cursor is a pointer to this context area
- It contains all information needed for processing the statement
- In PL/SQL, the context area is controlled by Cursor
- A cursor contains information on a select statement and the rows of data accessed by it.

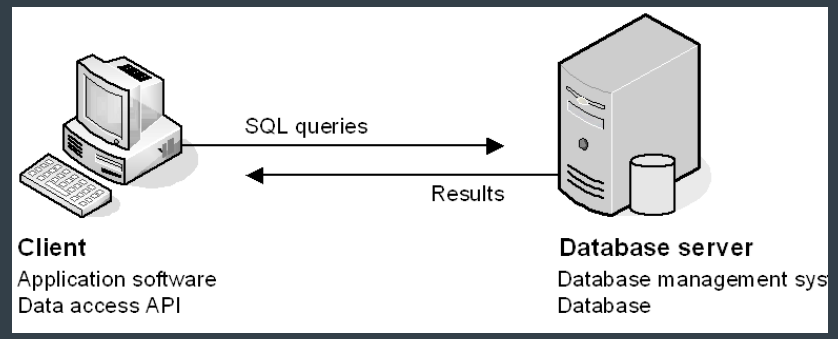
TYPES OF CURSORS

- **Implicit Cursors:** are automatically generated by Oracle while an SQL statement is executed (if you don't use an explicit cursor)
- **Explicit Cursors:** are defined by the programmers to gain more control over the context area
  - These cursors should be defined in the declaration section of the PL/SQL block
  - It is created on a SELECT statement which returns more than one row

DISCUSSION QUESTIONS

Where does anonymous PL/SQL is written and run?

Where does a stored procedure is saved and run?



LOOKING FORWARD

Read Chapters 3-8

- 13-15: PL/SQL

Quiz 4

Homework 4

Exam 2

THANK YOU

PART 2

Data Governance

QUESTION

Do you have old/unused data at home?

For example: a box of old VHS tapes, cassette tapes, or floppy disks, etc.

QUESTION

What do you need to successfully manage your data?

DATA GOVERNANCE…

… is a lot like taking care of your old VHS tapes.

- There's a lot of work to do to make it useful

Availability

Usability

Integrity

Security

DEFINITIONS

- Data Governance is the practice of organizing and implementing policies, procedures, and standards for the effective use of an organization's … information assets.
  - Anne Marie Smith, “Data Governance Maturity - An Overview”
- Data Stewardship: The formalization of accountability for the management of data resources
  - Bob Seiner, “The Data Stewardship Approach to Data Governance”

QUESTION

Do you have a memory of losing a data file?

Would you like to share the details?

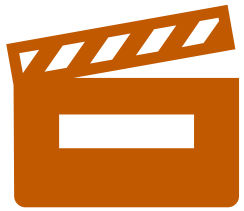
AVAILABILITY

Where is the data?

Is it connected?

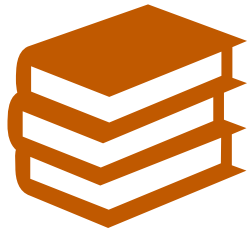
Is it restricted?

USABILITY



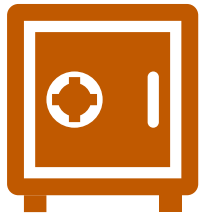
What format is used?

What language is used?



Is metadata available?

INTEGRITY



Are the data files free from corruption?

Is the code complete and functional?

Are the relations intact?

SECURITY

Are the data and code safe from hackers?

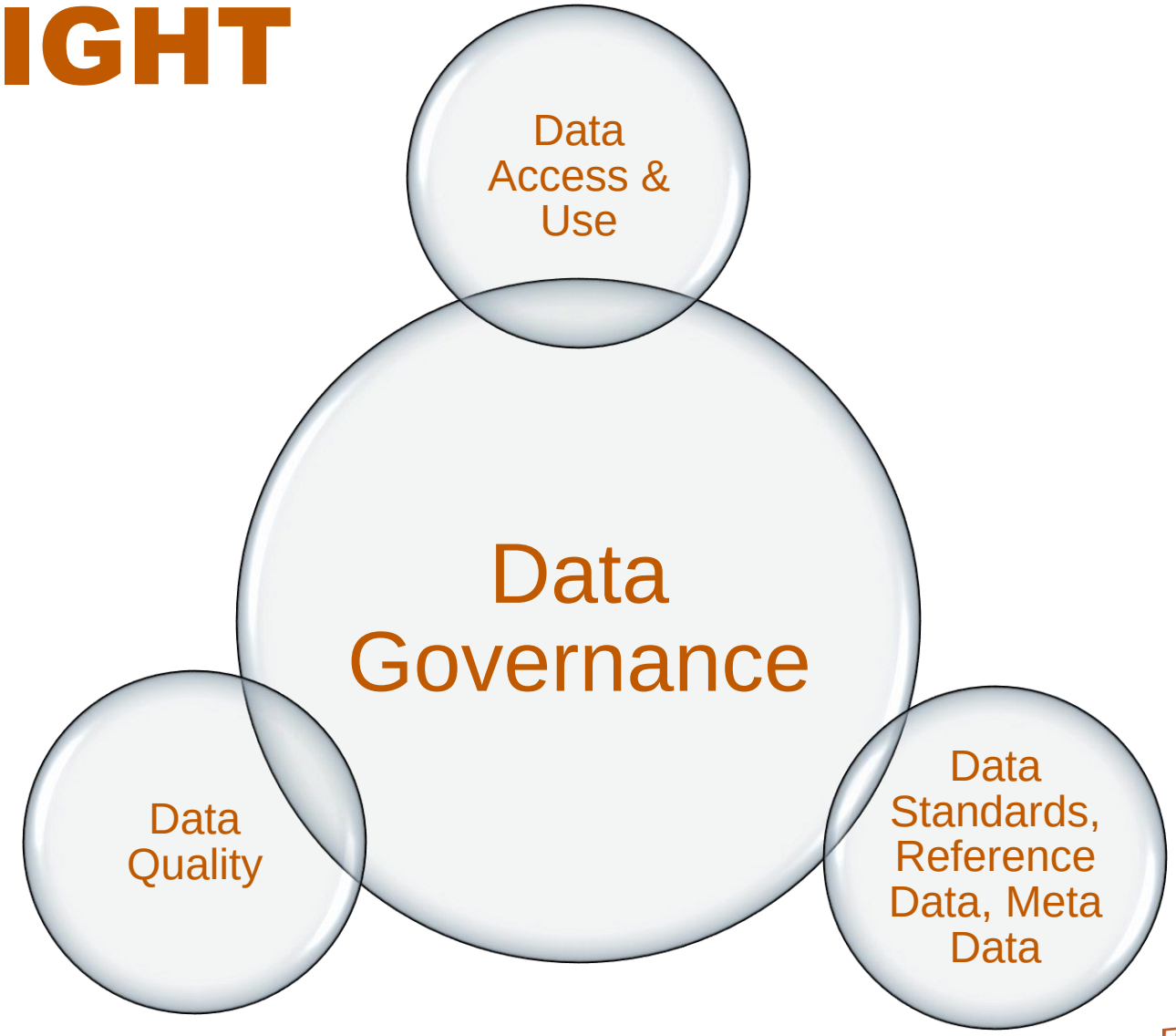
Are they safe from accidental changes by unauthorized users?

Are they future proof?

VALUE PROPOSITION

- Risk reduction
- Greater trust in data quality for use in decision making
- Cost reduction via improved efficiency
- Higher actual return on technology investments
- Increased value and utility of data assets

OVERSIGHT


Data Access & Use
Data Governance
Data Quality
Data Standards, Reference Data, Meta Data

QUESTION

What can you do for…

- Availability
- Usability
- Integrity
- Security

… of your data?

DELIVERABLES

- Establish a robust Data Stewardship Program to support Data Governance
- Develop, publish and manage enterprise-wide data definitions and semantics
- Identify, address, resolve and monitor data related issues
- Measure, monitor and communicate data quality parameters and issues

DELIVERABLES

- Publish and promote policies, procedures and rules, including those related to privacy, regulatory and contractual compliance
- Foster a culture and values that promote high quality data related practices and create zero tolerance for errors in mission-critical data
- Coordinate and facilitate the integration of data-driven business needs and priorities with IT requirements and constraints
- Provide business friendly Data Governance communication

THANK YOU

BACKUP SLIDES

PART 1

Chapter 13

PL/SQL Introduction

The syntax for an anonymous PL/SQL block

Optional

```
DECLARE
  declaration_statement_1;
  [declaration_statement_2;]...
BEGIN
  body_statement_1;
  [body_statement_2;]...
```

Optional

```
EXCEPTION
  WHEN OTHERS THEN
    exception_handling_statement_1;
    [exception_handling_statement_2;]...
END;
/
```

A script with an anonymous PL/SQL block (i.e. code snippet is nameless)

```
--CONNECT ap/ap;
SET SERVEROUTPUT ON;

DECLARE
  sum_balance_due_var NUMBER(9, 2);
BEGIN
  SELECT SUM(invoice_total - payment_total - credit_total)
  INTO sum_balance_due_var
  FROM invoices
  WHERE vendor_id = 95;
  IF sum_balance_due_var > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Balance due: $' || ROUND(sum_balance_due_var, 2));
  ELSE
    DBMS_OUTPUT.PUT_LINE('Balance paid in full');
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error occurred');
END;
/
```

Procedures for printing output to the screen

~~`DBMS_OUTPUT.ENABLE()`~~ --out of scope and only for old versions

`DBMS_OUTPUT.PUT(string)` = output string

`DBMS_OUTPUT.PUT_LINE(string)` = output string and move to next line

Commands for working with scripts

CONNECT

SET SERVEROUTPUT ON;

PL/SQL statements for controlling the flow

IF...ELSIF...ELSE

CASE...WHEN...ELSE

FOR...IN...LOOP

WHILE...LOOP

LOOP...EXIT WHEN

CURSOR...IS

EXECUTE IMMEDIATE

How to print data to the Script Output window

Oracle SQL Developer : C:\murach\oracle_sql\scripts\ch13\fig13-03.sql

File Edit View Navigate Run Source Team Tools Window Help

Connections

- ap
  - Tables (Filtered)
  - Views
  - Editioning Views
  - Indexes
  - Packages
  - Procedures
  - Functions
  - Queues
  - Queues Tables
  - Triggers
  - Crossedition Triggers
  - Types
  - Sequences
  - Materialized Views
  - Materialized View Logs
  - Synonyms
  - Public Synonyms
  - Database Links
  - Public Database Links
  - Directories

fig13-01.sql | fig13-03.sql

SQL Worksheet | History

Worksheet | Query Builder

```
CONNECT ap/ap;

SET SERVEROUTPUT ON;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Test SQL Developer');
END;
/
```

Script Output

Task completed in 0.006 seconds

```
Connected
anonymous block completed
Test SQL Developer

Connection created by CONNECT script command disconnected
```

SQL History

Line 5 Column 1 | Insert | Windows: CR

Declaring **VARIABLES**

The syntax for declaring a variable

```
variable_name_1 DATA_TYPE;
```

```
percent_difference NUMBER;
```

The syntax for declaring a variable with the same data type as a column

```
variable_name_1 table_name.column_name%TYPE;
```

```
max_invoice_total  invoices.invoice_total%TYPE;
```

The syntax for setting a variable to a selected value

```
SELECT column_1[, column_2]...
INTO variable_name_1[, variable_name_2]...
```

```
SELECT MAX(invoice_total), MIN(invoice_total)
INTO max_invoice_total, min_invoice_total
```

The syntax for setting a variable to a literal value or the result of an expression

```
variable_name := literal_value_or_expression
```

```
vendor_id_var      NUMBER := 95;
```

A SQL script that uses variables

```
DECLARE
  max_invoice_total  invoices.invoice_total%TYPE;
  min_invoice_total  invoices.invoice_total%TYPE;
  percent_difference NUMBER;
  count_invoice_id   NUMBER;
  vendor_id_var      NUMBER := 95;
BEGIN
  SELECT MAX(invoice_total), MIN(invoice_total), COUNT(invoice_id)
  INTO max_invoice_total, min_invoice_total, count_invoice_id
  FROM invoices WHERE vendor_id = vendor_id_var;

  percent_difference :=
      (max_invoice_total - min_invoice_total) /
       min_invoice_total * 100;

  DBMS_OUTPUT.PUT_LINE('Maximum invoice: $' ||
                       max_invoice_total);
  DBMS_OUTPUT.PUT_LINE('Minimum invoice: $' ||
                       min_invoice_total);
  DBMS_OUTPUT.PUT_LINE('Percent difference: %' ||
                       ROUND(percent_difference, 2));
  DBMS_OUTPUT.PUT_LINE('Number of invoices: ' ||
                       count_invoice_id);
END;
/
```

Handling **CONDITIONS**

The syntax of the IF statement

```
IF boolean_expression THEN
  statement_1;
  [statement_2;]...
[ELSIF boolean_expression THEN
  statement_1;
  [statement_2;]...]...
[ELSE
  statement_1;
  [statement_2;]...]
END IF;
```

NOTE spelling is ELSIF, not ELSEIF

A script that uses an IF statement

```
CONNECT ap/ap;
SET SERVEROUTPUT ON;

DECLARE
  first_invoice_due_date DATE;
BEGIN
  SELECT MIN(invoice_due_date)
  INTO first_invoice_due_date
  FROM invoices
  WHERE invoice_total - payment_total - credit_total > 0;

  IF first_invoice_due_date < SYSDATE() THEN
    DBMS_OUTPUT.PUT_LINE('Invoices overdue!');
  ELSIF first_invoice_due_date = SYSDATE() THEN
    DBMS_OUTPUT.PUT_LINE('Invoices are due today!');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No invoices are overdue.');
  END IF;
END;
/
```

The response from the system

```
Invoices overdue!
```

The syntax of the Simple CASE statement

```
CASE expression
  WHEN expression_value_1 THEN
    statement_1;
    [statement_2;]...
  [WHEN expression_value_2 THEN
    statement_1;
    [statement_2;]...]...
  [ELSE
    statement_1;
    [statement_2;]...]
END CASE;
```

A script that uses a Simple CASE statement

```
CONNECT ap/ap;
SET SERVEROUTPUT ON;

DECLARE
  terms_id_var NUMBER;
BEGIN
  SELECT terms_id INTO terms_id_var
  FROM invoices WHERE invoice_id = 4;

  CASE terms_id_var
    WHEN 1 THEN
      DBMS_OUTPUT.PUT_LINE('Net due 10 days');
    WHEN 2 THEN
      DBMS_OUTPUT.PUT_LINE('Net due 20 days');
    WHEN 3 THEN
      DBMS_OUTPUT.PUT_LINE('Net due 30 days');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Net due more than 30 days');
  END CASE;
END;
/
```

The syntax of a Searched CASE expression

```
CASE
  WHEN boolean_expression THEN
    statement_1;
    [statement_2;]...
  [WHEN boolean_expression THEN
    statement_1;
    [statement_2;]...]...
  [ELSE
    statement_1;
    [statement_2;]...]
END CASE;
```

No variable included in this kind of CASE

Repeating code with **LOOPS**

The syntax of the FOR loop

```
FOR counter_var IN [REVERSE]
      counter_start..counter_end LOOP
  statement_1;
  [statement_2;]...
END LOOP;
```

A FOR loop

```
FOR i IN 1..3 LOOP
  DBMS_OUTPUT.PUT_LINE('i: ' || i);
END LOOP;
```

The output for the loop

```
i: 1
i: 2
i: 3
```

The syntax for a WHILE loop

```
WHILE boolean_expression LOOP
  statement_1;
  [statement_2;]...
END LOOP;
```

A WHILE loop

```
i := 1;
WHILE i < 4 LOOP
  DBMS_OUTPUT.PUT_LINE('i: ' || i);
  i := i + 1;
END LOOP;
```

The output for the loop

```
i: 1
i: 2
i: 3
```

The syntax for a simple loop

```
LOOP
  statement_1;
  [statement_2;]...
  EXIT WHEN boolean_expression;
END LOOP;
```

A simple loop

```
i := 1;
LOOP
  DBMS_OUTPUT.PUT_LINE('i: ' || i);
  i := i + 1;
  EXIT WHEN i >= 4;
END LOOP;
```

The output for the loop

```
i: 1
i: 2
i: 3
```

CAUTION

- Stick to FOR or WHILE for simplicity's sake.
- Also beware using CONTINUE and CONTINUE WHEN. These are rarely use and EXIT WHEN can suffice.
- Also careful about write an ENDLESS LOOP which is…..?

Storing a list many values with Arrays, **BULK COLLECT**, & **CURSOR**

A script that uses a BULK COLLECT clause to populate a nested table

```
DECLARE
  TYPE names_table        IS TABLE OF VARCHAR2(40);
  vendor_names            names_table;
BEGIN
  SELECT vendor_name
  BULK COLLECT INTO vendor_names
  FROM vendors
  WHERE rownum < 4
  ORDER BY vendor_id;

  FOR i IN 1..vendor_names.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Vendor name: ' || vendor_names(i);
  END LOOP;
END;
/
```

Helpful when pulling a column (i.e., a list) of values

The response from the system

```
Vendor name: US Postal Service
Vendor name: National Information Data Ctr
Vendor name: Register of Copyrights
```

Practice

Update example to respond with first 10 vendor names.
Also add in vendor # (e.g. Vendor Name 1:) on each line like so:

```
Vendor name 1: US Postal Service
Vendor name 2: National Information Data Ctr
```

Cursors

```
DECLARE
   CURSOR invoices_cursor IS        --declare a variable for cursor
       SELECT invoice_id, invoice_total
       FROM invoices
       WHERE invoice_total - payment_total - credit_total > 0;

   invoice_row invoices%ROWTYPE;    --declare a variable for each row

BEGIN
   FOR invoice_row IN invoices_cursor LOOP  --for each row in the dataset

     . . . . . . .

   END LOOP;

END;
/
```

Helpful when pulling a table of values

	INVOICE_ID	INVOICE_TOTAL
1	3	20551.18
2	6	2312.2
3	8	1927.54
4	15	313.55
5	18	904.14
6	19	1962.13
7	30	17.5
8	34	10976.06
9	38	61.5
10	39	158
11	40	26.75
12	41	23.5
13	42	9.95
14	44	52.25

Handling **ERRORS**

A script that doesn't handle exceptions

```
CONNECT ap/ap;
INSERT INTO general_ledger_accounts VALUES (130, 'Cash');
```

The response from the system

```
SQL Error: ORA-00001: unique constraint
(AP.GL_ACCOUNT_DESCRIPTION_UQ) violated 00001. 00000 -
"unique constraint (%s.%s) violated"
*Cause:   An UPDATE or INSERT statement attempted to insert
          duplicate key. For Trusted Oracle configured in
          DBMS MAC mode, you may see this message if a
          duplicate entry exists at a different level.
*Action:  Either remove the unique restriction or do not
          insert the key.
```

The syntax of the EXCEPTION block

```
EXCEPTION
  WHEN most_specific_exception THEN
    statement_1;
    [statement_2;]...

  [WHEN less_specific_exception THEN
    statement_1;
    [statement_2;]...]...
```

An EXCEPTION block that handles exceptions

```
CONNECT ap/ap;
SET SERVEROUTPUT ON;

BEGIN
  INSERT INTO general_ledger_accounts VALUES (130, 'Cash');

  DBMS_OUTPUT.PUT_LINE('1 row inserted.');
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE(
        'You attempted to insert a duplicate value.');

  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(
        'An unexpected exception occurred.');
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

The response from the system

```
You attempted to insert a duplicate value.
```

A list of common exceptions

ORA Error	Exception
00001	DUP_VAL_ON_INDEX
01403	NO_DATA_FOUND
01476	ZERO_DIVIDE
01722	INVALID_NUMBER
06502	VALUE_ERROR

A script that will display an error if the object doesn't already exist

```
CONNECT ap/ap;
DROP TABLE test1;
CREATE TABLE test1 (test_id NUMBER);
```

The response from the system

```
Connected

Error starting at line 2 in command:
DROP TABLE test1
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:
CREATE TABLE succeeded.
```

Executing SQL PL/SQL using **EXECUTE IMMEDIATE**

The EXECUTE IMMEDIATE statement

```
EXECUTE IMMEDIATE 'sql_string'
```

A script that won't display an error

```
CONNECT ap/ap;

BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE test1';
EXCEPTION
  WHEN OTHERS THEN
    NULL;
END;
/

CREATE TABLE test1 (test_id NUMBER);
```

The response from the system

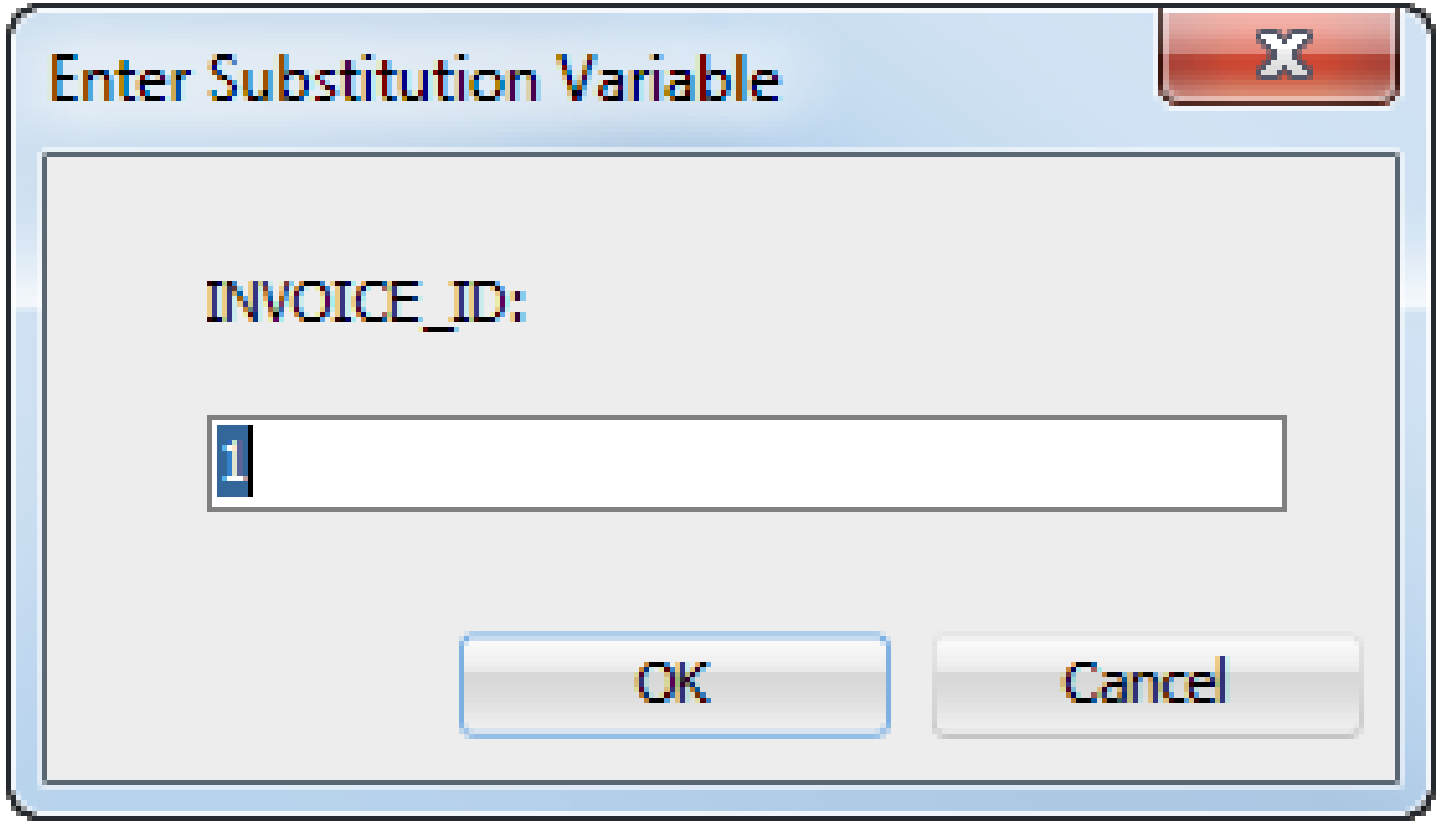
```
Connected
anonymous block completed
CREATE TABLE succeeded.
```

Prompting user entry with
SUBSTITUTION VARIABLES

A script that uses substitution variables

```
-- Use the VARIABLE keyword to declare a bind variable
VARIABLE invoice_id_value NUMBER;

-- Use a PL/SQL block to set the value of a bind variable
-- to the value of a substitution variable
BEGIN
  invoice_id_value := &invoice_id;
END;
/

-- Use a bind variable in a SELECT statement
SELECT invoice_id, invoice_number
FROM invoices
WHERE invoice_id = invoice_id_value;

-- Use a bind variable in another PL/SQL block
BEGIN
  DBMS_OUTPUT.PUT_LINE('invoice_id_value: ' || invoice_id_value);
END;
/
```

The dialog box for a substitution variable



The response from the system

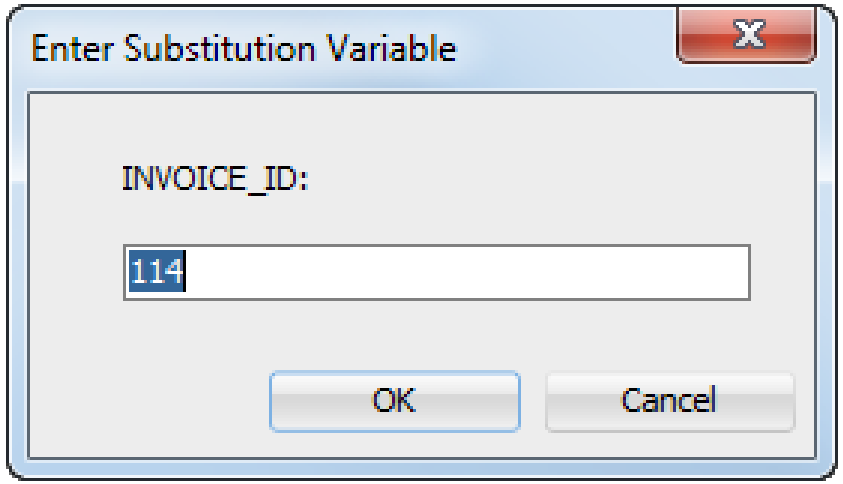
```
Connected
anonymous block completed
INVOICE_ID                INVOICE_NUMBER
------------------------- ------------------------------------
1                         QP58872

1 rows selected

anonymous block completed
invoice_id_value: 1
```

How to concatenate text/variables in **DYNAMIC SQL**

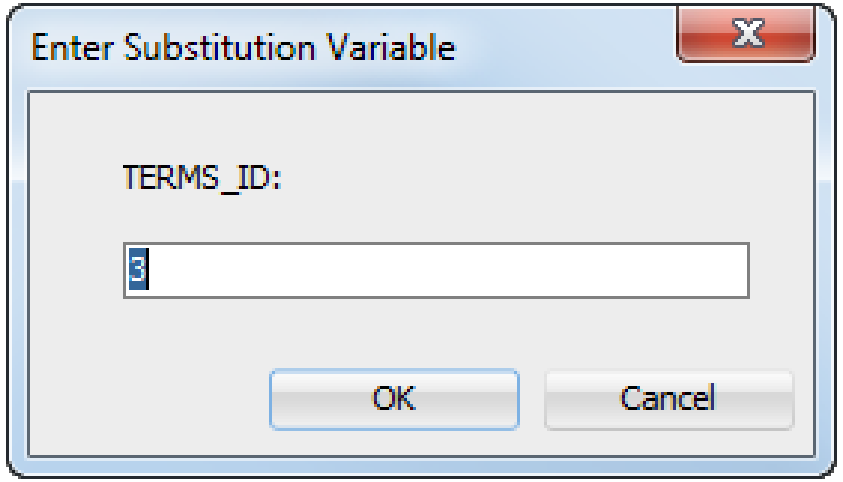
Dynamic SQL that updates a specified invoice

```
CONNECT ap/ap;

DECLARE
  invoice_id_var NUMBER;
  terms_id_var NUMBER;
  dynamic_sql VARCHAR2(400);
BEGIN
  invoice_id_var := &invoice_id;
  terms_id_var := &terms_id;

  dynamic_sql := 'UPDATE invoices ' ||
                 'SET terms_id = ' || terms_id_var || ' ' ||
                 'WHERE invoice_id = ' || invoice_id_var;

  EXECUTE IMMEDIATE dynamic_sql;
END;
/
```

The first dialog box for a substitution variable



The second dialog box for a substitution variable



The contents of the variable named dynamic_sql at runtime

```
UPDATE invoices SET terms_id = 3 WHERE invoice_id = 114
```

Appendix

The syntax for declaring a cursor

```
CURSOR cursor_name IS select_statement;
```

The syntax for declaring a variable for a row

```
row_variable_name table_name%ROWTYPE;
```

The syntax for getting a column value from a row variable

```
row_variable_name.column_name
```

A script that uses a cursor

```
DECLARE
  CURSOR invoices_cursor IS
    SELECT invoice_id, invoice_total  FROM invoices
    WHERE invoice_total - payment_total - credit_total > 0;

  invoice_row invoices%ROWTYPE;
BEGIN
  FOR invoice_row IN invoices_cursor LOOP

    IF (invoice_row.invoice_total > 1000) THEN
      UPDATE invoices
      SET credit_total = credit_total + (invoice_total * .1)
      WHERE invoice_id = invoice_row.invoice_id;
      DBMS_OUTPUT.PUT_LINE(
          '1 row updated where invoice_id = ' ||
          invoice_row.invoice_id);
    END IF;

  END LOOP;
END;
/
```

The response from the system

```
1 row updated where invoice_id = 3
1 row updated where invoice_id = 6
1 row updated where invoice_id = 8
1 row updated where invoice_id = 19
1 row updated where invoice_id = 34
1 row updated where invoice_id = 81
1 row updated where invoice_id = 88
1 row updated where invoice_id = 113
```

PART 2

Chapter 15

Stored Procedures and Functions

CREATE PROCEDURE syntax

```
CREATE [OR REPLACE] PROCEDURE proc_name
[(
    parameter_name_1 data_type
    [, parameter_name_2 data_type]...
)]

{IS | AS}

pl_sql_block
```

Example: sproc that updates a table

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
  invoice_number_param  VARCHAR2,
  credit_total_param    NUMBER
)

AS

BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;

  COMMIT;

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```

Example: sproc that updates a table

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
  invoice_number_param  VARCHAR2,
  credit_total_param    NUMBER
)

AS

BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;

  COMMIT;

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```

SQL statement that calls the sproc

```
CALL update_invoices_credit_total ('367447', 300);
```

Script that calls the sproc

```
BEGIN
 update_invoices_credit_total ('367447', 300);
END;
/
```

Script that passes parameters by name

```
BEGIN
 update_invoices_credit_total(
    credit_total_param => 300,
    invoice_number_param => '367447'
  );
END;
/
```

The syntax for declaring an optional parameter

```
parameter_name_1 data_type [DEFAULT default_value]
```

A statement that uses an optional parameter

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
  invoice_number_param VARCHAR2,
  credit_total_param   NUMBER    DEFAULT 100
)
AS
BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```

Practice

A statement that calls the stored procedure

```
CALL update_invoices_credit_total('367447', 200);
```

Another statement that calls the stored procedure

```
CALL update_invoices_credit_total('367447');
```

User –Defined FUNCTIONS

User Defined Functions

- A *user-defined function* (*UDF*), which can also be called a *stored function* or just a *function*, is an executable database object that contains a block of PL/SQL code.
- A *scalar-valued function* returns a single value of any data type.
- A function can accept input parameters that work like the input parameters for a stored procedure.
- A function always returns a value. You use the RETURN keyword to specify the value that's returned by the function.
- A function can't make changes to the database such as executing an INSERT, UPDATE, or DELETE statement.

The syntax for creating a function

```
CREATE [OR REPLACE] FUNCTION function_name
[(
    parameter_name_1 data_type
    [, parameter_name_2 data_type]...
)]
RETURN data_type
{IS | AS}
pl_sql_block
```

e.g. A function that returns a vendor ID

```
CREATE OR REPLACE FUNCTION get_vendor_id
(
   vendor_name_param VARCHAR2
)
RETURN NUMBER

AS
  vendor_id_var NUMBER;
BEGIN
  SELECT vendor_id
  INTO vendor_id_var
  FROM vendors
  WHERE vendor_name = vendor_name_param;

  RETURN vendor_id_var;
END;
/
```

e.g. A function that returns a vendor ID

```
CREATE OR REPLACE FUNCTION get_vendor_id
(
   vendor_name_param VARCHAR2
)
RETURN NUMBER
AS
  vendor_id_var NUMBER;
BEGIN
  SELECT vendor_id
  INTO vendor_id_var
  FROM vendors
  WHERE vendor_name = vendor_name_param;

  RETURN vendor_id_var;
END;
/
```

A SELECT statement that uses the function

```
SELECT invoice_number, invoice_total
FROM invoices
WHERE vendor_id = get_vendor_id('IBM')
```

The response from the system

	INVOICE_NUMBER	INVOICE_TOTAL
1	QP58872	116.54
2	Q545443	1083.58

A function that calculates balance due

```
CREATE OR REPLACE FUNCTION get_balance_due
(
   invoice_id_param NUMBER
)
RETURN NUMBER
AS
  balance_due_var NUMBER;
BEGIN
  SELECT invoice_total - payment_total - credit_total
      AS balance_due
  INTO balance_due_var
  FROM invoices
  WHERE invoice_id = invoice_id_param;

  RETURN balance_due_var;
END;
/
```

Statement that uses expression for balance_due

```
SELECT vendor_id, invoice_number,
       invoice_total - payment_total - credit_total AS balance_due
FROM invoices
WHERE vendor_id = 37;
```

Statement that calls function to get balance_due

```
SELECT vendor_id, invoice_number,
       get_balance_due(invoice_id) AS balance_due
FROM invoices
WHERE vendor_id = 37;
```

The response from the system

	VENDOR_ID	INVOICE_NUMBER	BALANCE_DUE
1	37	547479217	116
2	37	547480102	224
3	37	547481328	224

Advantages

1. Code is shorter & simpler
2. Easier to maintain - Logic to calculate balance due is stored in a single, centralized location. (i.e. not each individual query).

A statement that creates a function

```
CREATE FUNCTION get_sum_balance_due
(
   vendor_id_param NUMBER
)
RETURN NUMBER
AS
  sum_balance_due_var NUMBER;
BEGIN
  SELECT SUM(get_balance_due(invoice_id))
      AS sum_balance_due
  INTO sum_balance_due_var
  FROM invoices
  WHERE vendor_id = vendor_id_param;

  RETURN sum_balance_due_var;
END;
/
```

A statement that calls the function

```
SELECT vendor_id, invoice_number,
       get_balance_due(invoice_id) AS balance_due,
       get_sum_balance_due(vendor_id) AS sum_balance_due
FROM invoices
WHERE vendor_id = 37;
```

The response from the system

	VENDOR_ID	INVOICE_NUMBER	BALANCE_DUE	SUM_BALANCE_DUE
1	37	547479217	116	564
2	37	547480102	224	564
3	37	547481328	224	564

Appendix

The syntax for declaring parameters

```
parameter_name_1 [IN|OUT|IN OUT] data_type
```

A stored procedure that uses parameters

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
  invoice_number_param IN  VARCHAR2,
  credit_total_param   IN  NUMBER,
  update_count         OUT INTEGER
)
AS
BEGIN
  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;

  SELECT COUNT(*)
    INTO update_count
  FROM invoices
  WHERE invoice_number = invoice_number_param;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    SELECT 0 INTO update_count FROM dual;
    ROLLBACK;
END;
/
```

3rd parameter outputs into the variable declared when the procedure is executed

A script that calls the stored procedure

```
SET SERVEROUTPUT ON;

DECLARE
  row_count INTEGER;

BEGIN
  update_invoices_credit_total('367447', 200, row_count);
  DBMS_OUTPUT.PUT_LINE('row_count: ' || row_count);

END;
/
```

Practice – See #5

The syntax of the RAISE statement

```
RAISE exception_name
```

e.g. A procedure that raises a predefined exception

```
CREATE OR REPLACE PROCEDURE update_invoices_credit_total
(
  invoice_number_param VARCHAR2,
  credit_total_param   NUMBER
)
AS
BEGIN
  IF credit_total_param < 0 THEN
    RAISE VALUE_ERROR;
  END IF;

  UPDATE invoices
  SET credit_total = credit_total_param
  WHERE invoice_number = invoice_number_param;

  COMMIT;
END;
```

Note:

- This is a predefined error that can be used to identify invalid entries

A statement that calls the procedure

```
CALL update_invoices_credit_total('367447', -100);
```

e.g. The response from the system if not error catching

```
Error report:
SQL Error: ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "AP.UPDATE_INVOICES_CREDIT_TOTAL", line 9
```

e.g. Script that calls the procedure with error catching

```
SET SERVEROUTPUT ON;

BEGIN
  update_invoices_credit_total('367447', -100);
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Updating credit total to a negative is not allowed');
END;
/
```

The response from the system

```
Updating credit total to a negative is not allowed
```

The RAISE_APPLICATION_ERROR procedure

```
RAISE_APPLICATION_ERROR(error_number, error_message);
```

e.g. A statement that raises an application error

```
RAISE_APPLICATION_ERROR(-20001, 'Credit total may not be negative.');
```

Response if the error isn't caught

```
Error report:
SQL Error: ORA-20001: Credit total may not be negative.
ORA-06512: at "AP.UPDATE_INVOICES_CREDIT_TOTAL", line 10
```

e.g. A script that catches an application error

```
BEGIN
  update_invoices_credit_total('367447', -100);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(
        'An unknown exception occurred.');
END;
/
```

The response from the system

```
An unknown exception occurred.
```

Note:

- This is your own custom error message that we can add into the sproc. NOTE: you don't have the "catch" this error type in the EXCEPTION block
- While you don't have the "handle" this error type in the EXCEPTION block you can capture this with all OTHERS if you wish

Three statements that call the stored procedure

```
CALL insert_invoice(34, 'ZXA-080', '30-AUG-08', 14092.59,
                    0, 0, 3, '30-SEP-08', NULL);

CALL insert_invoice(34, 'ZXA-080', '30-AUG-08', 14092.59,
                    0, 0, 3, '30-SEP-08');

CALL insert_invoice(34, 'ZXA-080', '30-AUG-08',
                    14092.59);
```

The response for a successful insert

```
CALL insert_invoice(34, succeeded.
```

A statement that raises an error

```
CALL insert_invoice(34, 'ZXA-080', '30-AUG-08', -14092.59);
```

The response when a validation error occurs

```
Error report:
SQL Error: ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "AP.INSERT_INVOICE", line 20
```

A stored procedure that drops a table

```
CREATE OR REPLACE PROCEDURE drop_table
(
  table_name VARCHAR2
)
AS
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE ' || table_name;
EXCEPTION
  WHEN OTHERS THEN
    NULL;
END;
/
```

A statement that calls the stored procedure

```
CALL drop_table('test1');
```

The response from the system

```
CALL drop_table('test1') succeeded.
```

The syntax of the DROP PROCEDURE statement

```
DROP PROCEDURE procedure_name
```

e.g. A statement that creates a stored procedure

```
CREATE PROCEDURE clear_invoices_credit_total
(
  invoice_number_param  VARCHAR2
)
AS
BEGIN
  UPDATE invoices
  SET credit_total = 0
  WHERE invoice_number = invoice_number_param;

  COMMIT;
END;
/
```

A statement that drops a stored procedure

```
DROP PROCEDURE clear_invoices_credit_total
```

The syntax of the DROP FUNCTION statement

```
DROP FUNCTION function_name
```

A statement that drops a function

```
DROP FUNCTION get_sum_balance_due;
```