Introduction to PL/SQL

PL/SQL - Introduction

- Procedural extension allowing for modularity, variable declaration, loops and logical constructs.
- Allows for advanced error handling
- Communicates natively with other oracle database objects.
- Managed centrally within the Oracle database.

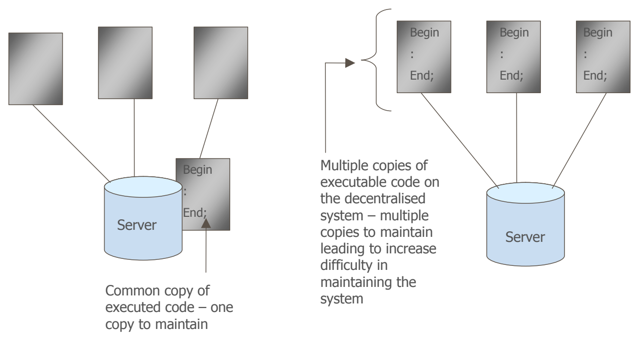
Other Databases

- All have procedural facilities
- SQL is not functionally complete
  - Lacks full facilities of a programming language
- So top up functionality by embedding SQL in a procedural language
- PL/SQL techniques are specific to Oracle
  - but procedures and functions can be ported to other systems

Why use PL/SQL

- Manage business rules – through *middle layer* application logic.
- Generate code for triggers
- Generate code for interface
- Enable database-centric client/server applications

Centralised VS De-centralised


Begin
:
End;
Server
Common copy of executed code – one copy to maintain
Multiple copies of executable code on the decentralised system – multiple copies to maintain leading to increase difficulty in maintaining the system
Begin
:
End;
Server

Advantages of using PL/SQL to access Oracle

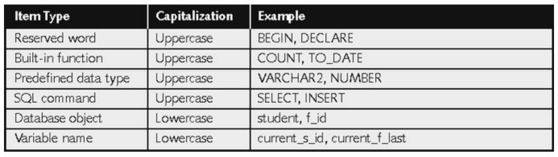
- PL/SQL is managed centrally within the database
- Code is managed by the DBA and execution privileges are managed in the same was as with other objects
- PL/SQL objects are first-class Oracle DB objects
- Easy to read
  - With modularity features and error handling

Centralised control

- Enables DBA to:
  - Specify rules in one place (as procedure, function, package in PL/SQL)
  - Force user access through the predefined PL/SQL so users cannot write their own procedural code and use this instead.
    - Define for instance security privileges giving users access to table(s) only through a particular procedure

Fundamentals of PL/SQL

- Full-featured programming language
- An interpreted language
- Type in editor, execute in SQL*Plus



Item Type	Capitalization	Example
Reserved word	Uppercase	BEGIN, DECLARE
Built-in function	Uppercase	COUNT, TO_DATE
Predefined data type	Uppercase	VARCHAR2, NUMBER
SQL command	Uppercase	SELECT, INSERT
Database object	Lowercase	student, f_id
Variable name	Lowercase	current_s_id, current_f_last

Table 4-1 PL/SQL command capitalization styles

Using PL/SQL as a programming language

- Permits all operations of standard programming languages e.g.
  - Conditions IF-THEN-ELSE-END IF;
  - Jumps GOTO
- Provides loops for controlling iteration
  - LOOP-EXIT; WHEN-END LOOP; FOR-END LOOP; WHILE-END LOOP
- Allows extraction of data into variables and its subsequent manipulation

Modules in PL/SQL

There are 4 types of modules in PL/SQL

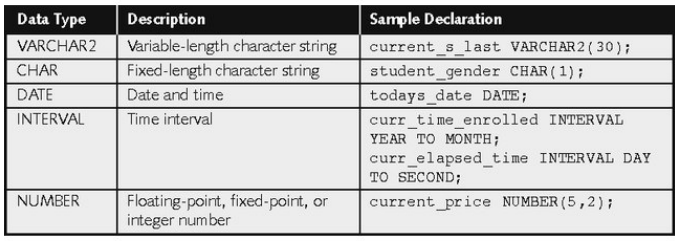
- *Procedures – series of statements may or may not return a value*
- *Functions – series of statements must return a single value*
- *Triggers – series of PL/SQL statements (actions) executing after an event has triggered a condition (ECA)*
- *Packages – collection of procedures and function that has 2 parts:*
  - *a listing and a body.*

Variables and Data Types

- Variables
  - Used to store numbers, character strings, dates, and other data values
  - Avoid using keywords, table names and column names as variable names
  - Must be declared with data type before use: *variable_name data_type_declaration*;

Scalar Data Types

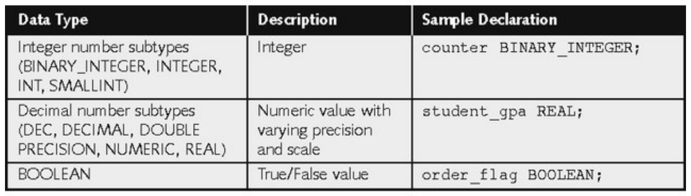
- Represent a single value



Data Type	Description	Sample Declaration
VARCHAR2	Variable-length character string	`current_s_last VARCHAR2(30);`
CHAR	Fixed-length character string	`student_gender CHAR(1);`
DATE	Date and time	`todays_date DATE;`
INTERVAL	Time interval	`curr_time_enrolled INTERVAL YEAR TO MONTH; curr_elapsed_time INTERVAL DAY TO SECOND;`
NUMBER	Floating-point, fixed-point, or integer number	`current_price NUMBER(5,2);`

Table 4-2 Scalar database data types

Scalar Data Types



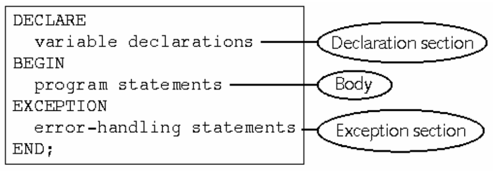
Data Type	Description	Sample Declaration
Integer number subtypes (BINARY_INTEGER, INTEGER, INT, SMALLINT)	Integer	`counter BINARY_INTEGER;`
Decimal number subtypes (DEC, DECIMAL, DOUBLE PRECISION, NUMERIC, REAL)	Numeric value with varying precision and scale	`student_gpa REAL;`
BOOLEAN	True/False value	`order_flag BOOLEAN;`

Table 4-3 General scalar data types

Composite and Reference Variables

- Composite variables
  - RECORD: contains multiple scalar values, similar to a table record
  - TABLE: tabular structure with multiple columns and rows
  - VARRAY: variable-sized array
- Reference variables
  - Directly reference a specific database field or record and assume the data type of the associated field or record
  - %TYPE: same data type as a database field
  - %ROWTYPE: same data type as a database record

PL/SQL Program Blocks

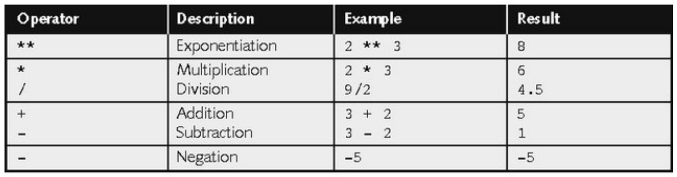


```
DECLARE
  variable declarations
BEGIN
  program statements
EXCEPTION
  error-handling statements
END;
```

Figure 4-1 Structure of a PL/SQL program block

- Comments:
  - Not executed by interpreter
  - Enclosed between /* and */
  - On one line beginning with --

Arithmetic Operators



Operator	Description	Example	Result
**	Exponentiation	2 ** 3	8
* /	Multiplication Division	2 * 3 9/2	6 4.5
+ -	Addition Subtraction	3 + 2 3 - 2	5 1
-	Negation	-5	-5

Table 4-5 PL/SQL arithmetic operators in describing order of precedence

Assignment Statements

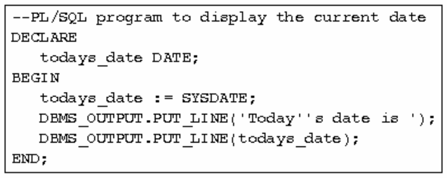
- Assigns a value to a variable

  variable_name := *value*;
- Value can be a literal:

  current_s_first_name := 'John';
- Value can be another variable:

  current_s_first_name := s_first_name;

Executing a PL/SQL Program in SQL*Plus



```
--PL/SQL program to display the current date
DECLARE
   todays_date DATE;
BEGIN
   todays_date := SYSDATE;
   DBMS_OUTPUT.PUT_LINE('Today''s date is ');
   DBMS_OUTPUT.PUT_LINE(todays_date);
END;
```

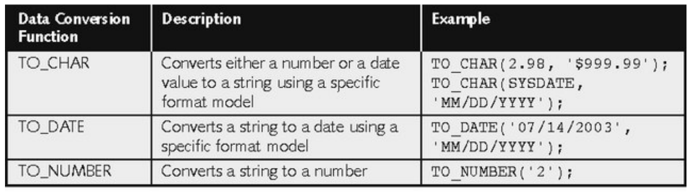
Figure 4-2 PL/SQL program commands

- Create program in text editor
- Paste into SQL*Plus window
- Press Enter, type / then enter to execute

Printing output

```
SET SERVEROUTPUT ON SIZE 4000

DECLARE
  my_name student.name%type;
BEGIN
    SELECT name INTO my_name FROM student
   WHERE student.sid=200;
    DBMS_OUTPUT.PUT_LINE('My name is ' ||
   my_name || '.');
 END;
```

PL/SQL Data Conversion Functions



Data Conversion Function	Description	Example
TO_CHAR	Converts either a number or a date value to a string using a specific format model	`TO_CHAR(2.98, '$999.99');` `TO_CHAR(SYSDATE, 'MM/DD/YYYY');`
TO_DATE	Converts a string to a date using a specific format model	`TO_DATE('07/14/2003', 'MM/DD/YYYY');`
TO_NUMBER	Converts a string to a number	`TO_NUMBER('2');`

Table 4-6 PL/SQL data conversion functions

Manipulating Character Strings with PL/SQL

- To concatenate two strings in PL/SQL, you use the double bar (||) operator:
  - *new_string* := *string1* || *string2*;
- To remove blank leading spaces use the LTRIM function:
  - *string := LTRIM(string_variable_name);*
- To remove blank trailing spaces use the RTRIM function:
  - *string := RTRIM(string_variable_name);*
- To find the number of characters in a character string use the LENGTH function:
  - *string_length := LENGTH(string_variable_name);*

Manipulating Character Strings with PL/SQL

- To change case, use UPPER, LOWER, INITCAP
- INSTR function searches a string for a specific substring:
  - *start_position := INSTR(original_string, substring);*
- SUBSTR function extracts a specific number of characters from a character string, starting at a given point:
  - *extracted_string := SUBSTR(string_variable, starting_point, number_of_characters);*

PL/SQL Decision Control Structures

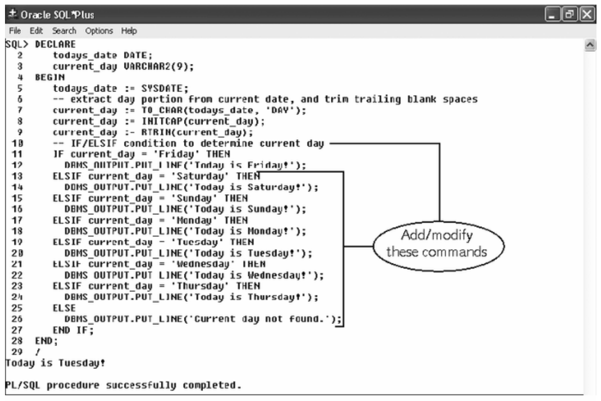
- Use IF/THEN structure to execute code if condition is true
  - IF *condition* THEN
    commands that execute if condition is TRUE;
    END IF;
- If condition evaluates to NULL it is considered false
- Use IF/THEN/ELSE to execute code if condition is true or false
  - IF *condition* THEN
    commands that execute if condition is TRUE;
    ELSE
    commands that execute if condition is FALSE;
    END IF;
- Can be nested – be sure to end nested statements

PL/SQL Decision Control Structures

- Use IF/ELSIF to evaluate many conditions:

```
IF condition1 THEN
   commands that execute if condition1 is TRUE;
 ELSIF condition2 THEN
   commands that execute if condition2 is TRUE;
 ELSIF condition3 THEN
   commands that execute if condition3 is TRUE;
 ...
 ELSE
   commands that execute if none of the
   conditions are TRUE;
 END IF;
```

IF/ELSIF Example



```
SQL> DECLARE
       todays_date DATE;
       current_day VARCHAR2(9);
     BEGIN
       todays_date := SYSDATE;
       -- extract day portion from current date, and trim trailing blank spaces
       current_day := TO_CHAR(todays_date, 'DAY');
       current_day := INITCAP(current_day);
       current_day := RTRIM(current_day);
       -- IF/ELSIF condition to determine current day
       IF current_day = 'Friday' THEN
         DBMS_OUTPUT.PUT_LINE('Today is Friday!');
       ELSIF current_day = 'Saturday' THEN
         DBMS_OUTPUT.PUT_LINE('Today is Saturday!');
       ELSIF current_day = 'Sunday' THEN
         DBMS_OUTPUT.PUT_LINE('Today is Sunday!');
       ELSIF current_day = 'Monday' THEN
         DBMS_OUTPUT.PUT_LINE('Today is Monday!');
       ELSIF current_day = 'Tuesday' THEN
         DBMS_OUTPUT.PUT_LINE('Today is Tuesday!');
       ELSIF current_day = 'Wednesday' THEN
         DBMS_OUTPUT.PUT_LINE('Today is Wednesday!');
       ELSIF current_day = 'Thursday' THEN
         DBMS_OUTPUT.PUT_LINE('Today is Thursday!');
       ELSE
         DBMS_OUTPUT.PUT_LINE('Current day not found.');
       END IF;
     END;
     /
Today is Tuesday!

PL/SQL procedure successfully completed.
```

Figure 4-17 Using an IF/ELSIF structure

Complex Conditions

- Created with logical operators AND, OR and NOT
- AND is evaluated before OR
- Use () to set precedence

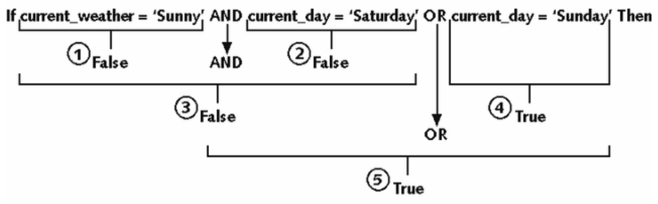

If current_weather = 'Sunny' AND current_day = 'Saturday' OR current_day = 'Sunday' Then
① False
AND
② False
③ False
OR
④ True
⑤ True

Figure 4-19 Evaluating AND and OR in an expression

Using SQL Queries in PL/SQL Programs

- Action queries can be used as in SQL*Plus
- May use variables in action queries
- DDL commands may not be used in PL/SQL

Loops

- Program structure that executes a series of program statements, and periodically evaluates an exit condition to determine if the loop should repeat or exit
- Pretest loop: evaluates the exit condition before any program commands execute
- Posttest loop: executes one or more program commands before the loop evaluates the exit condition for the first time
- PL/SQL has 5 loop structures

The LOOP...EXIT Loop

```
LOOP
  [program statements]
  IF condition THEN
    EXIT;
  END IF;
  [additional program statements]
END LOOP
```

The LOOP...EXIT WHEN Loop

```
LOOP
    program statements
    EXIT WHEN condition;
END LOOP;
```

The WHILE...LOOP

```
WHILE condition LOOP
    program statements
END LOOP;
```

The Numeric FOR Loop

```
FOR counter_variable IN start_value
  .. end_value
LOOP
    program statements
END LOOP;
```

Cursors

- Pointer to a memory location that the DBMS uses to process a SQL query
- Use to retrieve and manipulate database data

Using an Implicit Cursor

- Executing a SELECT query creates an implicit cursor
- To retrieve it into a variable use INTO:
  - SELECT *field1*, *field2*, ...
    INTO *variable1*, *variable2*, ...
    FROM *table1*, *table2*, ...
    WHERE *join_ conditions*
    AND
    search_condition_to_retrieve_1_record;
- Can only be used with queries that return exactly one record

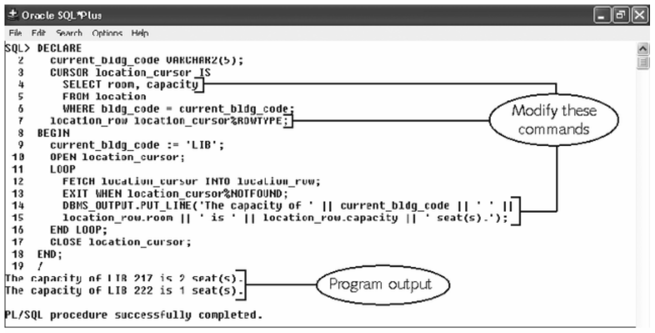
Explicit Cursor

- Use for queries that return multiple records or no records
- Must be explicitly declared and used

Using an Explicit Cursor

- Declare the cursor
  - *CURSOR cursor_name IS select_query;*
- Open the cursor
  - *OPEN cursor_name;*
- Fetch the data rows
  - *LOOP*
    FETCH cursor_name INTO variable_name(s);
    EXIT WHEN cursor_name%NOTFOUND;
- Close the cursor
  - *CLOSE cursor_name;*

Explicit Cursor with %ROWTYPE



```
SQL> DECLARE
  2    current_bldg_code VARCHAR2(5);
  3    CURSOR location_cursor IS
  4      SELECT room, capacity
  5      FROM location
  6      WHERE bldg_code = current_bldg_code;
  7    location_row location_cursor%ROWTYPE;
  8  BEGIN
  9    current_bldg_code := 'LIB';
 10    OPEN location_cursor;
 11    LOOP
 12      FETCH location_cursor INTO location_row;
 13      EXIT WHEN location_cursor%NOTFOUND;
 14      DBMS_OUTPUT.PUT_LINE('The capacity of ' || current_bldg_code || ' ' ||
 15      location_row.room || ' is ' || location_row.capacity || ' seat(s).');
 16    END LOOP;
 17    CLOSE location_cursor;
 18  END;
 19  /
The capacity of LIB 217 is 2 seat(s).
The capacity of LIB 222 is 1 seat(s).

PL/SQL procedure successfully completed.
```

Figure 4-31 Processing an explicit cursor using a %ROWTYPE variable

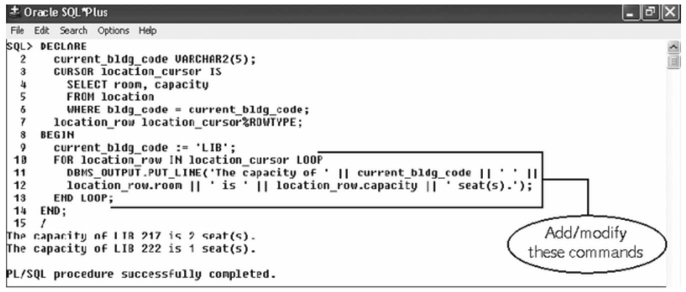
Cursor FOR Loop

- Automatically opens the cursor, fetches the records, then closes the cursor
- *FOR variable_name(s) IN cursor_name LOOP*

  processing commands

  END LOOP;
- Cursor variables cannot be used outside loop

Using Cursor FOR Loop



```
SQL> DECLARE
  2    current_bldg_code VARCHAR2(5);
  3    CURSOR location_cursor IS
  4      SELECT room, capacity
  5      FROM location
  6      WHERE bldg_code = current_bldg_code;
  7    location_row location_cursor%ROWTYPE;
  8  BEGIN
  9    current_bldg_code := 'LIB';
 10    FOR location_row IN location_cursor LOOP
 11      DBMS_OUTPUT.PUT_LINE('The capacity of ' || current_bldg_code || ' ' ||
 12      location_row.room || ' is ' || location_row.capacity || ' seat(s).');
 13    END LOOP;
 14  END;
 15  /
The capacity of LIB 217 is 2 seat(s).
The capacity of LIB 222 is 1 seat(s).

PL/SQL procedure successfully completed.
```

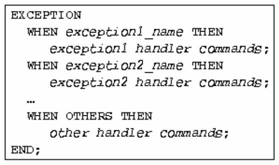
Figure 4-32 Processing an explicit cursor using a cursor FOR loop

Example using PLSQL

- Add a status column to student table
- Set status to FULL if the student takes 2 or more courses
- Set status to PART if the student takes less than 2 courses
- Print the results for the each of the update

Handling Runtime Errors in PL/SQL Programs

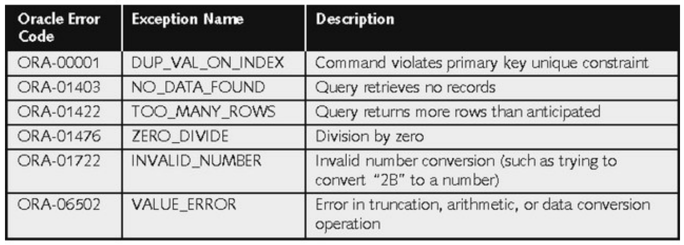
- Runtime errors cause exceptions
- Exception handlers exist to deal with different error situations
- Exceptions cause program control to fall to exception section where exception is handled



```
EXCEPTION
  WHEN exception1_name THEN
     exception1 handler commands;
  WHEN exception2_name THEN
     exception2 handler commands;
  …
  WHEN OTHERS THEN
     other handler commands;
END;
```

Figure 4-34 Exception handler syntax

Predefined Exceptions



Oracle Error Code	Exception Name	Description
ORA-00001	DUP_VAL_ON_INDEX	Command violates primary key unique constraint
ORA-01403	NO_DATA_FOUND	Query retrieves no records
ORA-01422	TOO_MANY_ROWS	Query returns more rows than anticipated
ORA-01476	ZERO_DIVIDE	Division by zero
ORA-01722	INVALID_NUMBER	Invalid number conversion (such as trying to convert "2B" to a number)
ORA-06502	VALUE_ERROR	Error in truncation, arithmetic, or data conversion operation

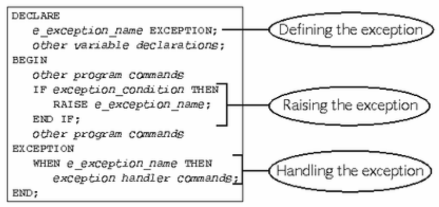
Table 4-10 Common PL/SQL predefined exceptions

Undefined Exceptions

- Less common errors
- Do not have predefined names
- Must declare your own name for the exception code in the declaration section

User-Defined Exceptions

- Not a real Oracle error
- Use to enforce business rules



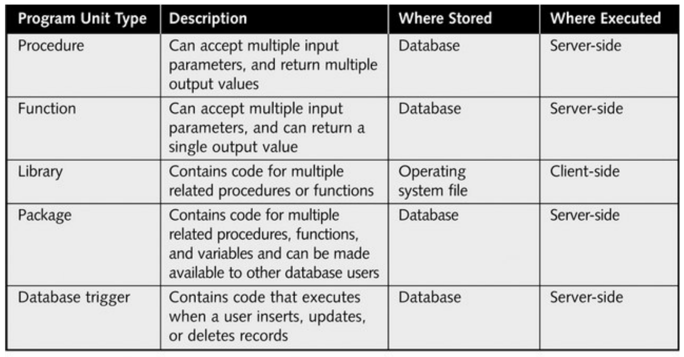
```
DECLARE
   e_exception_name EXCEPTION;
   other variable declarations;
BEGIN
   other program commands
   IF exception_condition THEN
      RAISE e_exception_name;
   END IF;
   other program commands
EXCEPTION
   WHEN e_exception_name THEN
      exception handler commands;
END;
```

Figure 4-40 General syntax for declaring, raising, and handling a user-defined exception

Overview of PL/SQL Stored Program Units

- Self-contained group of program statements that can be used within a larger program.
- Easier to conceptualize, design, and debug
- Save valuable programming time because you can reuse them in multiple database applications
- Other PL/SQL programs can reference them

Types of Program Units

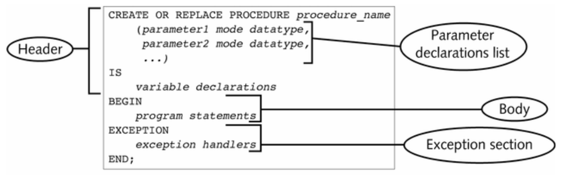


Program Unit Type	Description	Where Stored	Where Executed
Procedure	Can accept multiple input parameters, and return multiple output values	Database	Server-side
Function	Can accept multiple input parameters, and can return a single output value	Database	Server-side
Library	Contains code for multiple related procedures or functions	Operating system file	Client-side
Package	Contains code for multiple related procedures, functions, and variables and can be made available to other database users	Database	Server-side
Database trigger	Contains code that executes when a user inserts, updates, or deletes records	Database	Server-side

Table 9-1 Types of Oracle9*i* stored program units

Creating Stored Program Units

- **Procedure**: a program unit that can receive multiple input parameters and return multiple output values or return no output values
- **Function**: a program unit that can receive multiple input parameters, and always returns a single output value.



```
CREATE OR REPLACE PROCEDURE procedure_name
    (parameter1 mode datatype,
     parameter2 mode datatype,
     ...)
IS
    variable declarations
BEGIN
    program statements
EXCEPTION
    exception handlers
END;
```

Figure 9-9 Syntax to create a stored program unit procedure

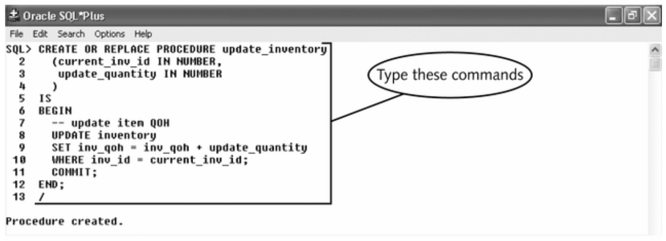
Parameter Declarations List

- Defines the parameters and declares their associated data types
- Enclosed in parentheses
- Separated by commas

Parameter Declarations List

- Parameter mode describes how the program unit can change the parameter value:
  - IN - specifies a parameter that is passed to the program unit as a read-only value that the program unit cannot change.
  - OUT - specifies a parameter that is a write-only value that can appear only on the left side of an assignment statement in the program unit
  - IN OUT - specifies a parameter that is passed to the program unit, and whose value can also be changed within the program unit

Creating a Stored Procedure in SQL*Plus



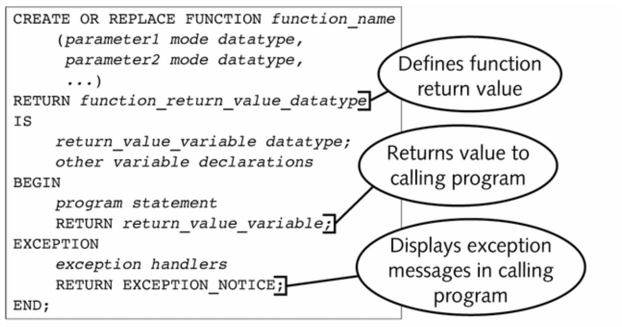
```
SQL> CREATE OR REPLACE PROCEDURE update_inventory
  2    (current_inv_id IN NUMBER,
  3     update_quantity IN NUMBER
  4    )
  5  IS
  6  BEGIN
  7    -- update item QOH
  8    UPDATE inventory
  9    SET inv_qoh = inv_qoh + update_quantity
 10    WHERE inv_id = current_inv_id;
 11    COMMIT;
 12  END;
 13  /

Procedure created.
```

Figure 9-10 Creating a stored procedure in SQL*Plus

Calling a Stored Procedure

- From SQL*Plus command line:
  - *EXECUTE procedure_name (parameter1_value, parameter2_value, ...);*
- From PL/SQL program:
  - Omit execute command
- Passing parameters (see Figure 9-13)

Creating a Stored Program Unit Function

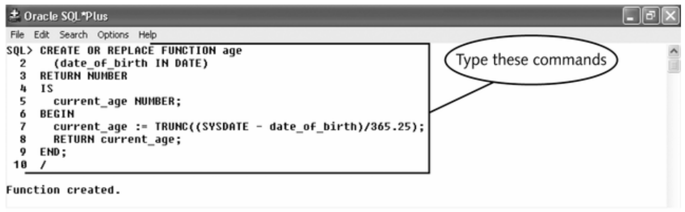

Defines function return value
Returns value to calling program
Displays exception messages in calling program

```
CREATE OR REPLACE FUNCTION function_name
     (parameter1 mode datatype,
      parameter2 mode datatype,
      ...)
RETURN function_return_value_datatype
IS
     return_value_variable datatype;
     other variable declarations
BEGIN
     program statement
     RETURN return_value_variable;
EXCEPTION
     exception handlers
     RETURN EXCEPTION_NOTICE;
END;
```

Figure 9-16 Syntax to create a stored procedure unit function

Creating a Stored Program Unit Function

- Last command in function must be RETURN

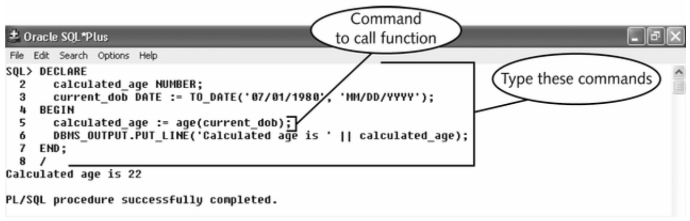


```
SQL> CREATE OR REPLACE FUNCTION age
  2    (date_of_birth IN DATE)
  3  RETURN NUMBER
  4  IS
  5    current_age NUMBER;
  6  BEGIN
  7    current_age := TRUNC((SYSDATE - date_of_birth)/365.25);
  8    RETURN current_age;
  9  END;
 10  /

Function created.
```

Figure 9-17 Creating a stored program unit function

Calling a Function

- *variable_name := function_name(parameter1, parameter2, ...);*



```
SQL> DECLARE
  2    calculated_age NUMBER;
  3    current_dob DATE := TO_DATE('07/01/1980', 'MM/DD/YYYY');
  4  BEGIN
  5    calculated_age := age(current_dob);
  6    DBMS_OUTPUT.PUT_LINE('Calculated age is ' || calculated_age);
  7  END;
  8  /
Calculated age is 22

PL/SQL procedure successfully completed.
```

Figure 9-18 Calling a function from an anonymous PL/SQL program

Debugging PL/SQL Programs

- Syntax error:
  - Command does not follow the guidelines of the programming language
  - Generates compiler or interpreter error messages
- Logic error:
  - Program runs but results in an incorrect result
  - Caused by mistake in program

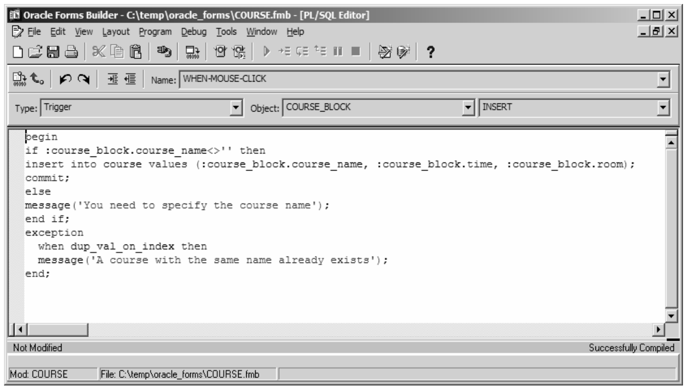
Finding and Fixing Syntax Errors

- Interpreter flags the line number and character location of syntax errors
- If error message appears and the flagged line appears correct, the error usually occurs on program lines *preceding* the flagged line
- Comment out program lines to look for hidden errors
- One error (such as missing semicolon) may cause more – fix one error at a time

Finding and Fixing Logic Errors

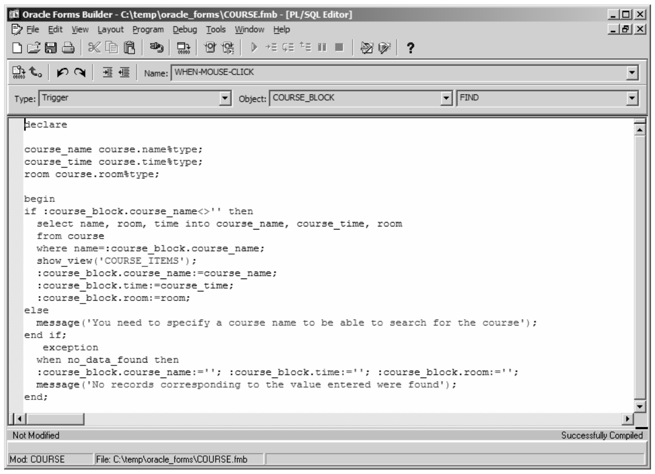
- Locate logic errors by viewing variable values during program execution
- There is no SQL*Plus debugger
- Use DBMS_OUTPUT statements to print variable values

Adding PL/SQL code


Oracle Forms Builder - C:\temp\oracle_forms\COURSE.fmb - [PL/SQL Editor]
File Edit View Layout Program Debug Tools Window Help
Not Modified
Successfully Compiled
Mod: COURSE
File: C:\temp\oracle_forms\COURSE.fmb

Name: WHEN-MOUSE-CLICK
Type: Trigger   Object: COURSE_BLOCK   INSERT

```
begin
if :course_block.course_name<>'' then
insert into course values (:course_block.course_name, :course_block.time, :course_block.room);
commit;
else
message('You need to specify the course name');
end if;
exception
  when dup_val_on_index then
  message('A course with the same name already exists');
end;
```

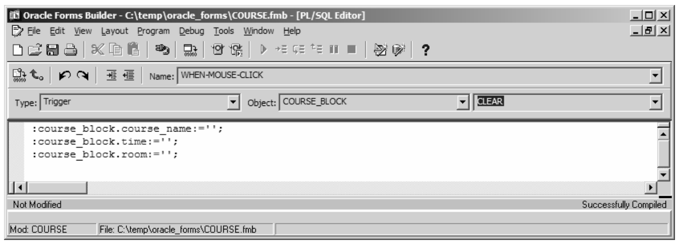

Oracle Forms Builder - C:\temp\oracle_forms\COURSE.fmb - [PL/SQL Editor]
File Edit View Layout Program Debug Tools Window Help
Not Modified
Successfully Compiled
Mod: COURSE
File: C:\temp\oracle_forms\COURSE.fmb

Name: WHEN-MOUSE-CLICK
Type: Trigger   Object: COURSE_BLOCK   FIND

```
declare

course_name course.name%type;
course_time course.time%type;
room course.room%type;

begin
if :course_block.course_name<>'' then
  select name, room, time into course_name, course_time, room
  from course
  where name=:course_block.course_name;
  show_view('COURSE_ITEMS');
  :course_block.course_name:=course_name;
  :course_block.time:=course_time;
  :course_block.room:=room;
else
  message('You need to specify a course name to be able to search for the course');
end if;
   exception
  when no_data_found then
  :course_block.course_name:=''; :course_block.time:=''; :course_block.room:='';
  message('No records corresponding to the value entered were found');
end;
```

Adding PL/SQL code

- Create triggers for each of the buttons
- Add PL/SQL code to the triggers

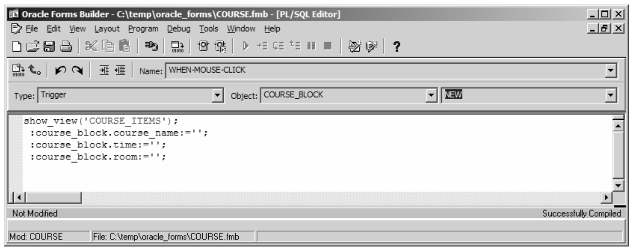

Oracle Forms Builder - C:\temp\oracle_forms\COURSE.fmb - [PL/SQL Editor]
File Edit View Layout Program Debug Tools Window Help
Not Modified
Successfully Compiled
Mod: COURSE
File: C:\temp\oracle_forms\COURSE.fmb

Name: WHEN-MOUSE-CLICK
Type: Trigger   Object: COURSE_BLOCK   CLEAR

```
 :course_block.course_name:='';
 :course_block.time:='';
 :course_block.room:='';
```

Adding PL/SQL code

- Create triggers for each of the buttons
- Add PL/SQL code to the triggers



Oracle Forms Builder - C:\temp\oracle_forms\COURSE.fmb - [PL/SQL Editor]
File Edit View Layout Program Debug Tools Window Help

Name: WHEN-MOUSE-CLICK

Type: Trigger Object: COURSE_BLOCK NEW

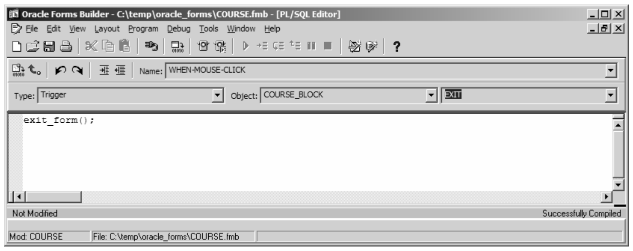
```
show_view('COURSE_ITEMS');
 :course_block.course_name:='';
 :course_block.time:='';
 :course_block.room:='';
```

Not Modified Successfully Compiled

Mod: COURSE File: C:\temp\oracle_forms\COURSE.fmb

Adding PL/SQL code

- Create triggers for each of the buttons
- Add PL/SQL code to the triggers



Oracle Forms Builder - C:\temp\oracle_forms\COURSE.fmb - [PL/SQL Editor]
File Edit View Layout Program Debug Tools Window Help

Name: WHEN-MOUSE-CLICK

Type: Trigger Object: COURSE_BLOCK EXIT

```
exit_form();
```

Not Modified Successfully Compiled

Mod: COURSE File: C:\temp\oracle_forms\COURSE.fmb