Review of PL/SQL

Block Structure for Anonymous PL/SQL Blocks

- `DECLARE` (optional)
  - Declare PL/SQL objects to be used within this block.
- `BEGIN` (mandatory)
  - Define the executable statements.
- `EXCEPTION` (optional)
  - Define the actions that take place if an error or exception arises.
- `END;` (mandatory)

Anonymous Blocks

Anonymous blocks do not have names. You declare them at the point in an application where they are to be run, and they are passed to the PL/SQL engine for execution at run time.

- The section between the keywords `DECLARE` and `BEGIN` is referred to as the declaration section. In the declaration section, you define the PL/SQL objects such as variables, constants, cursors, and user-defined exceptions that you want to reference within the block. The `DECLARE` keyword is optional if you do not declare any PL/SQL objects.
- The `BEGIN` and `END` keywords are mandatory and enclose the body of actions to be performed. This section is referred to as the executable section of the block.
- The section between `EXCEPTION` and `END` is referred to as the exception section. The exception section traps error conditions. In it, you define actions to take if a specified condition arises. The exception section is optional.

The keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not followed by semicolons, but `END` and all other PL/SQL statements do require semicolons.

Declaring PL/SQL Variables

- **Syntax:**

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

- **Examples:**

```
Declare
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13)  := 'Atlanta';
  c_ comm         CONSTANT NUMBER := 1400;
  v_count         BINARY_INTEGER := 0;
  v_valid         BOOLEAN NOT NULL := TRUE;
```


Declaring PL/SQL Variables

You need to declare all PL/SQL identifiers within the declaration section before referencing them within the PL/SQL block. You have the option to assign an initial value. You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, you must be sure to declare them separately in a previous statement.

In the syntax,

Identifier	Is the name of the variable
CONSTANT	Constrains the variable so that its value cannot change; constants must be initialized.
datatype	Is a scalar, composite, reference, or LOB data type (This course covers only scalar and composite data types.)
NOT NULL	Constrains the variable so that it must contain a value; NOT NULL variables must be initialized.
expr	Is any PL/SQL expression that can be a literal, another variable, or an expression involving operators and functions

Declaring Variables with the %TYPE Attribute

Examples:

```
...
  v_ename                employees.last_name%TYPE;
  v_balance              NUMBER(7,2);
  v_min_balance          v_balance%TYPE := 10;
...
```


Declaring Variables with the %TYPE Attribute

Declare variables to store the name of an employee.

```
...
v_ename          employees.last_name%TYPE;
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10.

```
...
v_balance        NUMBER(7,2);
v_min_balance    v_balance%TYPE := 10;
...
```

A `NOT NULL` column constraint does not apply to variables declared using `%TYPE`. Therefore, if you declare a variable using the `%TYPE` attribute and a database column defined as `NOT NULL`, then you can assign the `NULL` value to the variable.

Creating a PL/SQL Record

Declare variables to store the name, job, and salary of a new employee.

Example:

```
...
  TYPE emp_record_type IS RECORD
    (ename    VARCHAR2(25),
     job         VARCHAR2(10),
     sal         NUMBER(8,2));
  emp_record     emp_record_type;
...
```


Creating a PL/SQL Record

Field declarations are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the data type first and then declare an identifier using that data type.

The following example shows that you can use the `%TYPE` attribute to specify a field data type:

```
DECLARE
  TYPE emp_record_type IS RECORD
    (empid  NUMBER(6) NOT NULL := 100,
     ename  employees.last_name%TYPE,
     job    employees.job_id%TYPE);
  emp_record     emp_record_type;
...
```

Note: You can add the `NOT NULL` constraint to any field declaration to prevent the assigning of nulls to that field. Remember that fields declared as `NOT NULL` must be initialized.

%ROWTYPE Attribute

Examples:

- **Declare a variable to store the same information about a department as is stored in the DEPARTMENTS table.**

```
dept_record    departments%ROWTYPE;
```

- **Declare a variable to store the same information about an employee as is stored in the EMPLOYEES table.**

```
emp_record     employees%ROWTYPE;
```


Examples

The first declaration in the slide creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID.

The second declaration in the slide creates a record with the same field names and field data types as a row in the EMPLOYEES table. The fields are EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID.

In the following example, you select column values into a record named item_record.

```
DECLARE
   job_record  jobs%ROWTYPE;
   ...
BEGIN
   SELECT * INTO job_record
   FROM   jobs
   WHERE  ...
```

Creating a PL/SQL Table

```
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
  ename_table     ename_table_type;
  hiredate_table  hiredate_table_type;
BEGIN
  ename_table(1) := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
      INSERT INTO ...
    ...
END;
```

ORACLE

Creating a PL/SQL Table

There are no predefined data types for PL/SQL tables, as there are for scalar variables. Therefore, you must create the data type first and then declare an identifier using that data type.

Referencing a PL/SQL Table

Syntax

```
pl/sql_table_name(primary_key_value)
```

In this syntax, `primary_key_value` belongs to the `BINARY_INTEGER` type.

Reference the third row in a PL/SQL table `ENAME_TABLE`.

```
ename_table(3) ...
```

The magnitude range of a `BINARY_INTEGER` is –2147483647 to 2147483647. The primary key value can therefore be negative. Indexing need not start with 1.

Note: The `table.EXISTS(i)` statement returns `TRUE` if at least one row with index `i` is returned. Use the `EXISTS` statement to prevent an error that is raised in reference to a nonexistent table element.

SELECT Statements in PL/SQL

The INTO clause is mandatory.

Example:

```
DECLARE
  v_deptid    NUMBER(4);
  v_loc       NUMBER(4);
BEGIN
  SELECT  department_id, location_id
  INTO    v_deptno, v_loc
  FROM    departments
  WHERE   department_name = 'Sales';
...
END;
```


INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables to hold the values that SQL returns from the SELECT clause. You must give one variable for each item selected, and the order of variables must correspond to the items selected.

You use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return One and Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of Embedded SQL, for which the following rule applies:

Queries must return one and only one row. More than one row or no row generates an error.

PL/SQL deals with these errors by raising standard exceptions, which you can trap in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. You should code SELECT statements to return a single row.

Inserting Data

Add new employee information to the `EMPLOYEES` table.

Example:

```
DECLARE
  v_empid  employees.employee_id%TYPE;
BEGIN
  SELECT  employees_seq.NEXTVAL
  INTO    v_empno
  FROM    dual;
  INSERT INTO  employees(employee_id, last_name,
                         job_id, department_id)
  VALUES(v_empno, 'HARDING', 'PU_CLERK', 30);
END;
```


Inserting Data

- Use SQL functions, such as `USER` and `SYSDATE`.
- Generate primary key values by using database sequences.
- Derive values in the PL/SQL block.
- Add column default values.

Note: There is no possibility for ambiguity with identifiers and column names in the `INSERT` statement. Any identifier in the `INSERT` clause must be a database column name.

Updating Data

Increase the salary of all employees in the `EMPLOYEES` table who are purchasing clerks.

Example:

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 2000;
BEGIN
  UPDATE  employees
  SET     salary = salary + v_sal_increase
  WHERE   job_id = 'PU_CLERK';
END;
```


Updating Data

There may be ambiguity in the `SET` clause of the `UPDATE` statement because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable.

Remember that the `WHERE` clause is used to determine which rows are affected. If no rows are modified, no error occurs (unlike the `SELECT` statement in PL/SQL).

Note: PL/SQL variable assignments always use `:=` and SQL column assignments always use `=`.. Remember that if column names and identifier names are identical in the `WHERE` clause, the Oracle server looks to the database first for the name.

Deleting Data

Delete rows that belong to department 190 from the `EMPLOYEES` table.

Example:

```
DECLARE
  v_deptid   employees.department_id%TYPE := 190;
BEGIN
  DELETE FROM employees
  WHERE department_id = v_deptid;
END;
```


Deleting Data

Delete a specific job:

```
DECLARE
  v_jobid    jobs.job_id%TYPE := 'PR_REP';
BEGIN
  DELETE FROM jobs
  WHERE job_id = v_jobid;
END;
```

COMMIT and ROLLBACK Statements

- Initiate a transaction with the first DML command to follow a COMMIT or ROLLBACK statement.
- Use COMMIT and ROLLBACK SQL statements to terminate a transaction explicitly.

Controlling Transactions

You control the logic of transactions with COMMIT and ROLLBACK SQL statements, rendering some groups of database changes permanent while discarding others. As with the Oracle server, data manipulation language (DML) transactions start at the first command to follow a COMMIT or ROLLBACK and end on the next successful COMMIT or ROLLBACK. These actions may occur within a PL/SQL block or as a result of events in the host environment. A COMMIT ends the current transaction by making all pending changes to the database permanent.

Syntax

```
COMMIT [WORK];
ROLLBACK [WORK];
```

In this syntax, WORK is for compliance with ANSI standards.

Note: The transaction control commands are all valid within PL/SQL, although the host environment may place some restriction on their use.

You can also include explicit locking commands (such as LOCK TABLE and SELECT ... FOR UPDATE) in a block. They stay in effect until the end of the transaction. Also, one PL/SQL block does not necessarily imply one transaction.

SQL Cursor Attributes

Using SQL cursor attributes, you can test the outcome of your SQL statements.

`SQL%ROWCOUNT`	**Number of rows affected by the most recent SQL statement (an integer value)**
`SQL%FOUND`	**Boolean attribute that evaluates to `TRUE` if the most recent SQL statement affects one or more rows**
`SQL%NOTFOUND`	**Boolean attribute that evaluates to `TRUE` if the most recent SQL statement does not affect any rows**
`SQL%ISOPEN`	**Boolean attribute that always evaluates to `FALSE` because PL/SQL closes implicit cursors immediately after they are executed**

SQL Cursor Attributes

SQL cursor attributes enable you to evaluate what happened when the implicit cursor was last used. You use these attributes in PL/SQL statements such as functions. You cannot use them in SQL statements.

You can use the `SQL%ROWCOUNT`, `SQL%FOUND`, `SQL%NOTFOUND`, and `SQL%ISOPEN` attributes in the exception section of a block to gather information about the execution of a DML statement. In PL/SQL, a DML statement that does not change any rows is not seen as an error condition, whereas the `SELECT` statement will return an exception if it cannot locate any rows.

IF, THEN, and ELSIF Statements

For a given value entered, return a calculated value.

Example:

```
. . .
IF v_start > 100 THEN
  v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
  v_start := 0.5 * v_start;
ELSE
  v_start := 0.1 * v_start;
END IF;
. . .
```


IF, THEN, and ELSIF Statements

When possible, use the ELSIF clause instead of nesting IF statements. The code is easier to read and understand, and the logic is clearly identified. If the action in the ELSE clause consists purely of another IF statement, it is more convenient to use the ELSIF clause. This makes the code clearer by removing the need for nested END IFs at the end of each further set of conditions and actions.

Example

```
IF condition1 THEN
  statement1;
ELSIF condition2 THEN
  statement2;
ELSIF condition3 THEN
  statement3;
END IF;
```

The statement in the slide is further defined as follows:

For a given value entered, return a calculated value. If the entered value is over 100, then the calculated value is two times the entered value. If the entered value is between 50 and 100, then the calculated value is 50% of the starting value. If the entered value is less than 50, then the calculated value is 10% of the starting value.

Note: Any arithmetic expression containing null values evaluates to null.

Basic Loop

Example:

```
DECLARE
  v_ordid     order_items.order_id%TYPE := 101;
  v_counter   NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```


Basic Loop

The basic loop example shown in the slide is defined as follows:

Insert the first 10 new line items for order number 101.

Note: A basic loop enables execution of its statements at least once, even if the condition has been met upon entering the loop.

`FOR` Loop

Insert the first 10 new line items for order number 101.

Example:

```
DECLARE
  v_ordid     order_items.order_id%TYPE := 101;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, i);
  END LOOP;
END;
```

ORACLE

`FOR` Loop

The slide shows a `FOR` loop that inserts 10 rows into the `order_items` table.

WHILE Loop

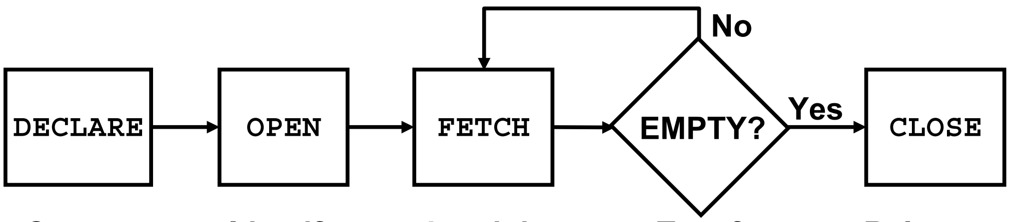
Example:

```
ACCEPT p_price PROMPT 'Enter the price of the item: '
ACCEPT p_itemtot -
 PROMPT 'Enter the maximum total for purchase of item: '
DECLARE
...
v_qty                 NUMBER(8) := 1;
v_running_total       NUMBER(7,2) := 0;
BEGIN
  ...
  WHILE v_running_total < &p_itemtot LOOP
    ...
  v_qty := v_qty + 1;
  v_running_total := v_qty * &p_price;
  END LOOP;
...
```


WHILE Loop

In the example in the slide, the quantity increases with each iteration of the loop until the quantity is no longer less than the maximum price allowed for spending on the item.

Controlling Explicit Cursors



- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to *FETCH* if rows are found.
- Release the active set.

Explicit Cursors

Controlling Explicit Cursors Using Four Commands

1. Declare the cursor by naming it and defining the structure of the query to be performed within it.
2. Open the cursor. The `OPEN` statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor. The `FETCH` statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore, each fetch accesses a different row returned by the query. In the flow diagram in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise, it closes the cursor.
4. Close the cursor. The `CLOSE` statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Declaring the Cursor

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR c2 IS
    SELECT *
    FROM   departments
    WHERE  department_id = 10;
BEGIN
  ...
```


Explicit Cursor Declaration

Retrieve the employees one by one.

```
DECLARE
  v_empid   employees.employee_id%TYPE;
  v_ename   employees.last_name%TYPE;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
...
```

Note: You can reference variables in the query, but you must declare them before the CURSOR statement.

Opening the Cursor

Syntax:

```
OPEN cursor_name;
```

- **Open the cursor to execute the query and identify the active set.**
- **If the query returns no rows, no exception is raised.**
- **Use cursor attributes to test the outcome after a fetch.**

`OPEN` Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax, `cursor_name` is the name of the previously declared cursor.

`OPEN` is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information
2. Parses the `SELECT` statement
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the `OPEN` statement is executed. Rather, the `FETCH` statement retrieves the rows.
5. Positions the pointer just before the first row in the active set

Note: If the query returns no rows when the cursor is opened, then PL/SQL does not raise an exception. However, you can test the cursor's status after a fetch.

For cursors declared by using the `FOR UPDATE` clause, the `OPEN` statement also locks those rows.

Fetching Data from the Cursor

Examples:

```
FETCH c1 INTO v_empid, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```


FETCH Statement

You use the FETCH statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their data types must be compatible. Retrieve the first 10 employees one by one:

```
DECLARE
  v_empid  employees.employee_id%TYPE;
  v_ename  employees.last_name%TYPE;
  i          NUMBER := 1;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  OPEN c1;
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename;
    ...
  END LOOP;
END;
```

Closing the Cursor

Syntax:

```
CLOSE   cursor_name;
```

- **Close the cursor after completing the processing of the rows.**
- **Reopen the cursor, if required.**
- **Do not attempt to fetch data from a cursor after it has been closed.**

CLOSE Statement

The CLOSE statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore, you can establish an active set several times.

In the syntax, cursor_name is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor after it has been closed, or the INVALID_CURSOR exception will be raised.

Note: The CLOSE statement releases the context area. Although it is possible to terminate the PL/SQL block without closing cursors, you should always close any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter field. By default, the maximum number of OPEN_CURSORS is 50.

```
...
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename; ...
  END LOOP;
  CLOSE c1;
END;
```

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
`%ISOPEN`	`BOOLEAN`	Evaluates to `TRUE` if the cursor is open
`%NOTFOUND`	`BOOLEAN`	Evaluates to `TRUE` if the most recent fetch does not return a row
`%FOUND`	`BOOLEAN`	Evaluates to `TRUE` if the most recent fetch returns a row; complement of `%NOTFOUND`
`%ROWCOUNT`	`NUMBER`	Evaluates to the total number of rows returned so far

Explicit Cursor Attributes

As with implicit cursors, there are four attributes for obtaining status information about a cursor. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a DML statement.

Note: Do not reference cursor attributes directly in a SQL statement.

Cursor FOR Loops

Retrieve employees one by one until there are no more left.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  FOR emp_record IN c1 LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.employee_id = 134 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```


Cursor FOR Loops

A cursor `FOR` loop processes rows in an explicit cursor. The cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched.

FOR UPDATE Clause

Retrieve the orders for amounts over $1,000 that were processed today.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT customer_id, order_id
    FROM   orders
    WHERE  order_date = SYSDATE
      AND  order_total > 1000.00
    ORDER BY customer_id
    FOR UPDATE NOWAIT;
```


FOR UPDATE Clause

If the database server cannot acquire the locks on the rows it needs in a `SELECT FOR UPDATE`, then it waits indefinitely. You can use the `NOWAIT` clause in the `SELECT FOR UPDATE` statement and test for the error code that returns due to failure to acquire the locks in a loop. Therefore, you can retry opening the cursor *n* times before terminating the PL/SQL block.

If you intend to update or delete rows by using the `WHERE CURRENT OF` clause, you must specify a column name in the `FOR UPDATE OF` clause.

If you have a large table, you can achieve better performance by using the `LOCK TABLE` statement to lock all rows in the table. However, when using `LOCK TABLE`, you cannot use the `WHERE CURRENT OF` clause and must use the notation `WHERE` *`column = identifier`*.

WHERE CURRENT OF Clause

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT salary FROM employees
    FOR UPDATE OF salary NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
    ...
  END LOOP;
  COMMIT;
END;
```


WHERE CURRENT OF Clause

You can update rows based on criteria from a cursor.

Additionally, you can write your `DELETE` or `UPDATE` statement to contain the `WHERE CURRENT OF cursor_name` clause to refer to the latest row processed by the `FETCH` statement. When you use this clause, the cursor you reference must exist and must contain the `FOR UPDATE` clause in the cursor query; otherwise, you get an error. This clause enables you to apply updates and deletes to the currently addressed row without the need to explicitly reference the `ROWID` pseudocolumn.

Trapping Predefined Oracle Server Errors

- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
  - `NO_DATA_FOUND`
  - `TOO_MANY_ROWS`
  - `INVALID_CURSOR`
  - `ZERO_DIVIDE`
  - `DUP_VAL_ON_INDEX`

Trapping Predefined Oracle Server Errors

Trap a predefined Oracle server error by referencing its standard name within the corresponding exception-handling routine.

Note: PL/SQL declares predefined exceptions in the `STANDARD` package.
It is a good idea to always consider the `NO_DATA_FOUND` and `TOO_MANY_ROWS` exceptions, which are the most common.

Trapping Predefined Oracle Server Errors: Example

Syntax:

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```


Trapping Predefined Oracle Server Exceptions: Example

In the example in the slide, a message is printed out to the user for each exception. Only one exception is raised and handled at any time.

Non-Predefined Error

Trap for Oracle server error number –2292, which is an integrity constraint violation.

```
DECLARE
  e_products_invalid EXCEPTION;
  PRAGMA EXCEPTION_INIT (
             e_products_invalid, -2292);
  v_message VARCHAR2(50);
BEGIN
. . .
EXCEPTION
  WHEN e_products_invalid THEN
    :g_message := 'Product ID
             specified is not valid.';
. . .
END;
```


Trapping a Non-Predefined Oracle Server Exception

1. Declare the name for the exception within the declarative section.
   Syntax

   ```
   exception   EXCEPTION;
   ```

   In this syntax, *exception* is the name of the exception.
2. Associate the declared exception with the standard Oracle server error number, using the `PRAGMA EXCEPTION_INIT` statement.
   Syntax

   ```
   PRAGMA EXCEPTION_INIT(exception, error_number);
   ```

   In this syntax:

exception	Is the previously declared exception
error_number	Is a standard Oracle server error number
3. Reference the declared exception within the corresponding exception-handling routine.
   In the slide example: If there is product in stock, halt processing and print a message to the user.

User-Defined Exceptions

Example:

```
[DECLARE]
  e_amount_remaining EXCEPTION;
. . .
BEGIN
. . .
  RAISE e_amount_remaining;
. . .
EXCEPTION
  WHEN e_amount_remaining THEN
    :g_message := 'There is still an amount
                    in stock.';
. . .
END;
```

1
2
3

ORACLE

Trapping User-Defined Exceptions

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name for the user-defined exception within the declarative section.
   Syntax: *exception* EXCEPTION;
   where: *exception* Is the name of the exception
2. Use the RAISE statement to raise the exception explicitly within the executable section.
   Syntax: RAISE *exception*;
   where: *exception* Is the previously declared exception
3. Reference the declared exception within the corresponding exception-handling routine.

In the slide example: This customer has a business rule that states that a product cannot be removed from its database if there is any inventory left in stock for this product. Because there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a DELETE on the PRODUCT_INFORMATION table, the block queries the INVENTORIES table to see whether there is any stock for the product in question. If there is stock, raise an exception.

Note: Use the RAISE statement by itself within an exception handler to raise the same exception back to the calling environment.

RAISE_APPLICATION_ERROR Procedure

Syntax:

```
raise_application_error (error_number,
            message[, {TRUE | FALSE}]);
```

- **Enables you to issue user-defined error messages from stored subprograms**
- **Is called from an executing stored subprogram only**

RAISE_APPLICATION_ERROR Procedure

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax, *error_number* is a user-specified number for the exception between –20000 and –20999. The *message* is the user-specified message for the exception. It is a character string that is up to 2,048 bytes long.

TRUE | FALSE is an optional Boolean parameter. If TRUE, the error is placed on the stack of previous errors. If FALSE (the default), the error replaces all previous errors.

Example:

```
 ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20201,
     'Manager is not a valid employee.');
END;
```

`RAISE_APPLICATION_ERROR` Procedure

- Is used in two different places:
  - `Executable` section
  - `Exception` section
- Returns error conditions to the user in a manner consistent with other Oracle server errors

ORACLE

`RAISE_APPLICATION_ERROR` Procedure: Example

```
...
DELETE FROM employees
WHERE  manager_id = v_mgr;
IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR(-20202,
    'This is not a valid manager');
END IF;
...
```