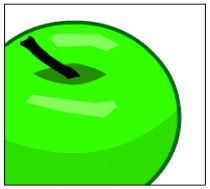
Retrieving Data in PL/SQL

Tell Me/Show Me



SQL Statements in PL/SQL

You can use the following kinds of SQL statements in PL/SQL:

- `SELECT` to retrieve data from the database.
- DML statements, such as `INSERT`, `UPDATE`, and `DELETE` to make changes to rows in the database.
- Transaction control statements, such as `COMMIT`, `ROLLBACK`, or `SAVEPOINT`. You use transaction control statements to make the changes to the database permanent or to discard them. Transaction control statements are covered later in the course.

This lesson covers `SELECT` statements.

Tell Me/Show Me

SQL Statements in PL/SQL (continued)

You cannot use DDL and DCL directly in PL/SQL.

Statement Type	Examples
DDL	`CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`
DCL	`GRANT`, `REVOKE`

PL/SQL does not directly support data definition language (DDL) statements, such as `CREATE TABLE`, `ALTER TABLE`, or `DROP TABLE` and DCL statements such as `GRANT` and `REVOKE`.

You cannot directly execute DDL and DCL statements because they are constructed and executed at run time. That is, they are dynamic. Static SQL statements are statements that are fixed at the time a program is compiled.

Tell Me/Show Me

SELECT Statements in PL/SQL

Retrieve data from the database with a SELECT statement.

Syntax:

```
SELECT   select_list
  INTO   {variable_name[, variable_name]...
         | record_name}
  FROM   table
 [WHERE condition];
```

Tell Me/Show Me

SELECT Statements in PL/SQL (continued)

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

```
DECLARE
  v_country_name wf_countries.country_name%TYPE;
BEGIN
  SELECT country_name INTO v_country_name
    FROM wf_countries  WHERE country_id= 359;
  DBMS_OUTPUT.PUT_LINE(' The country name is :
                        '||v_country_name);
END;
```

Tell Me/Show Me

Retrieving Data in PL/SQL

Retrieve `hire_date` and `salary` for the specified employee.

Example:

```
DECLARE
  v_emp_hiredate   employees.hire_date%TYPE;
  v_emp_salary     employees.salary%TYPE;
BEGIN
  SELECT     hire_date, salary
    INTO     v_emp_hiredate, v_emp_salary
    FROM     employees
    WHERE    employee_id = 100;
  DBMS_OUTPUT.PUT_LINE('Hiredate is: ' || v_emp_hiredate
                       || ' and Salary is: '
                       || v_emp_salary);
END;
```

Tell Me/Show Me

Retrieving Data in PL/SQL (continued)

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: queries must return exactly one row. A query that returns more than one row or no rows generates an error. You learn about error handling later in the course.

```
DECLARE
  v_salary employees.salary%TYPE;
BEGIN
  SELECT salary INTO v_salary
    FROM employees;
  DBMS_OUTPUT.PUT_LINE(' Salary is : ' || v_salary);
END;
```

```
ORA-01422: exact fetch returns more than requested number of rows
```

Tell Me/Show Me

Retrieving Data in PL/SQL (continued)

Return the sum of the salaries for all the employees in the specified department.

```
DECLARE
  v_sum_sal  NUMBER(10,2);
  v_deptno   NUMBER NOT NULL := 60;
BEGIN
  SELECT SUM(salary)  -- group function
    INTO v_sum_sal FROM employees
    WHERE  department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE ('The sum of salary is '
                        || v_sum_sal);
END;
```

Tell Me/Show Me

Guidelines for Retrieving Data in PL/SQL

- Terminate each SQL statement with a semicolon (;).
- Every value retrieved must be stored in a variable using the INTO clause.
- The WHERE clause is optional and can contain input variables, constants, literals, or PL/SQL expressions. However, you should fetch only one row and the usage of the WHERE clause is therefore needed in nearly all cases.
- Specify the same number of variables in the INTO clause as database columns in the SELECT clause. Be sure that they correspond positionally and that their data types are compatible.
- Declare the receiving variables using %TYPE.

Tell Me/Show Me

Guidelines for Naming Conventions

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables.

```
DECLARE
  v_hire_date      employees.hire_date%TYPE;
  employee_id      employees.employee_id%TYPE := 176;
BEGIN
  SELECT       hire_date
    INTO       v_hire_date
    FROM       employees
    WHERE      employee_id = employee_id;
END;
```

```
ORA-01422: exact fetch returns more than requested number of
```

This example raises an unhandled run-time exception because in the `WHERE` clause, the PL/SQL variable name is the same as that of the database column name in the `employees` table.

Tell Me/Show Me

Guidelines for Naming Conventions (continued)

What is deleted in the following PL/SQL block?

```
DECLARE
  last_name VARCHAR2(25) := 'King';
BEGIN
  DELETE FROM emp_dup WHERE last_name = last_name;
END;
```

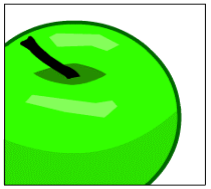
Tell Me/Show Me

Guidelines for Naming Conventions (continued)

- Use a naming convention to avoid ambiguity in the `WHERE` clause.
- Avoid using database column names as identifiers.
- Errors can occur during execution because PL/SQL checks the database first for a column in the table.
- The names of local variables and formal parameters take precedence over the names of database *tables (in a PL/SQL statement)*.
- The names of database table *columns* take precedence over the names of local variables.

Manipulating Data in PL/SQL

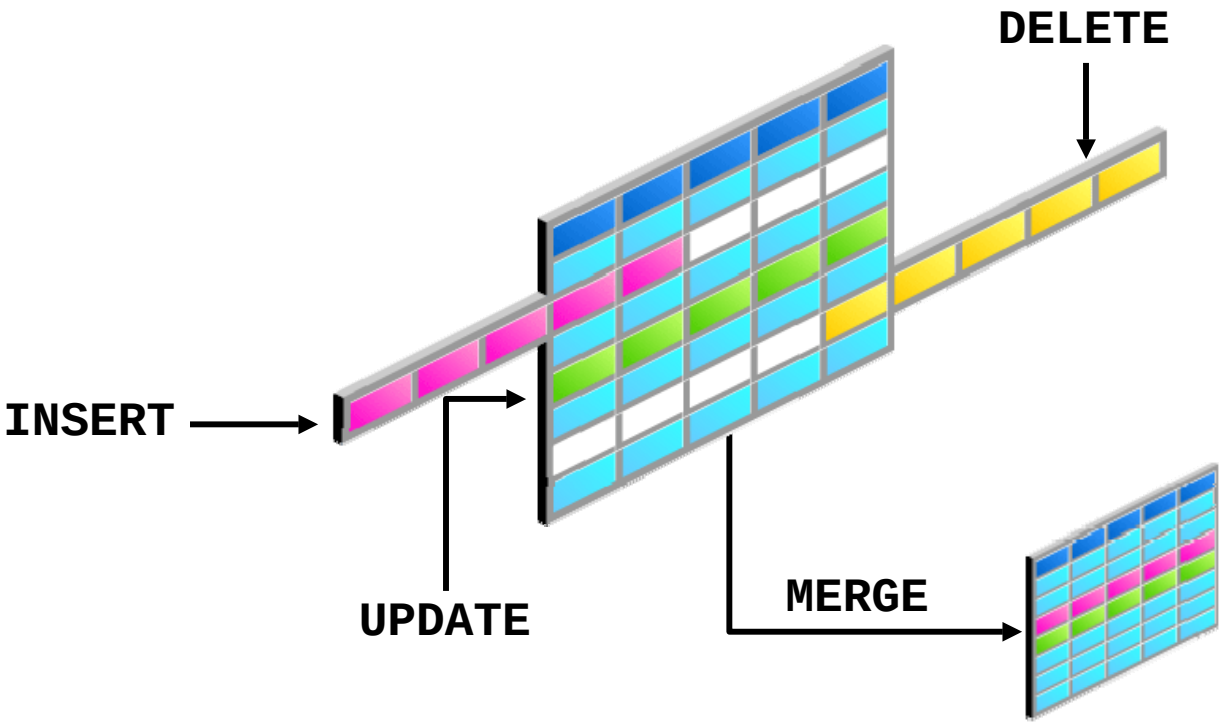
Tell Me/Show Me



Manipulating Data Using PL/SQL

Make changes to data by using DML commands within your PLSQL block:

- INSERT
- UPDATE
- DELETE
- MERGE



Tell Me/Show Me

Manipulating Data Using PL/SQL (continued)

- You manipulate data in the database by using the DML commands.
- You can issue the DML commands—`INSERT`, `UPDATE`, `DELETE`, and `MERGE`—without restriction in PL/SQL. Row locks (and table locks) are released by including `COMMIT` or `ROLLBACK` statements in the PL/SQL code.
  - The `INSERT` statement adds new rows to the table.
  - The `UPDATE` statement modifies existing rows in the table.
  - The `DELETE` statement removes rows from the table.
  - The `MERGE` statement selects rows from one table to update and/or insert into another table. The decision whether to update or insert into the target table is based on a condition in the `ON` clause.
- **Note:** `MERGE` is a deterministic statement—that is, you cannot update the same row of the target table multiple times in the same `MERGE` statement. You must have `INSERT` and `UPDATE` object privileges in the target table and the `SELECT` privilege in the source table.

Tell Me/Show Me

Inserting Data

The INSERT statement adds new row(s) to a table.

Example: Add new employee information to the COPY_EMP table.

```
BEGIN
  INSERT INTO copy_emp
     (employee_id, first_name, last_name, email,
      hire_date, job_id, salary)
   VALUES (99, 'Ruth', 'Cores',
           'RCORES', SYSDATE, 'AD_ASST', 4000);
END;
```

One new row is added to the COPY_EMP table.

Tell Me/Show Me

Updating Data

The UPDATE statement modifies existing row(s) in a table.

Example: Increase the salary of all employees who are stock clerks.

```
DECLARE
  v_sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE      copy_emp
    SET       salary = salary + v_sal_increase
    WHERE     job_id = 'ST_CLERK';
END;
```

Tell Me/Show Me

Deleting Data

The DELETE statement removes row(s) from a table.

Example: Delete rows that belong to department 10 from the COPY_EMP table.

```
DECLARE
  v_deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   copy_emp
    WHERE  department_id = v_deptno;
END;
```

Tell Me/Show Me

Merging Rows

The MERGE statement selects rows from one table to update and/or insert into another table. Insert or update rows in the copy_emp table to match the employees table.

```
BEGIN
  MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = c.employee_id)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name     = e.first_name,
       c.last_name      = e.last_name,
       c.email          = e.email,
       . . .
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id, e.first_name, …e.department_id);
END;
```

Tell Me/Show Me

Getting Information From a Cursor

Look again at the DELETE statement in this PL/SQL block.

```
DECLARE
  v_deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   copy_emp
    WHERE  department_id = v_deptno;
END;
```

It would be useful to know how many COPY_EMP rows were deleted by this statement.

To obtain this information, we need to understand cursors.

Tell Me/Show Me

What is a Cursor?

Every time an SQL statement is about to be executed, the Oracle server allocates a private memory area to store the SQL statement and the data that it uses. This memory area is called an implicit cursor.

Because this memory area is automatically managed by the Oracle server, you have no direct control over it. However, you can use predefined PL/SQL variables, called implicit cursor attributes, to find out how many rows were processed by the SQL statement.

Tell Me/Show Me

Implicit and Explicit Cursors

There are two types of cursors:

- Implicit cursors: Defined automatically by Oracle for all SQL data manipulation statements, and for queries that return only one row. An implicit cursor is always automatically named “SQL.”
- Explicit cursors: Defined by the PL/SQL programmer for queries that return more than one row. (Covered in a later lesson.)

Tell Me/Show Me

Cursor Attributes for Implicit Cursors

Cursor attributes are automatically declared variables that allow you to evaluate what happened when a cursor was last used. Attributes for implicit cursors are prefaced with "SQL." Use these attributes in PL/SQL statements, but not in SQL statements. Using cursor attributes, you can test the outcome of your SQL statements.

SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement returned at least one row
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement did not return even one row
SQL%ROWCOUNT	An integer value that represents the number of rows affected by the most recent SQL statement

Tell Me/Show Me

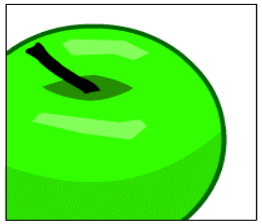
Using Implicit Cursor Attributes: Example 1

Delete rows that have the specified employee ID from the `copy_emp` table. Print the number of rows deleted.

```
DECLARE
  v_deptno copy_emp.department_id%TYPE := 50;
BEGIN
  DELETE FROM copy_emp
    WHERE department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||
                       ' rows deleted.');
END;
```

Using Transaction Control Statements

Tell Me/Show Me



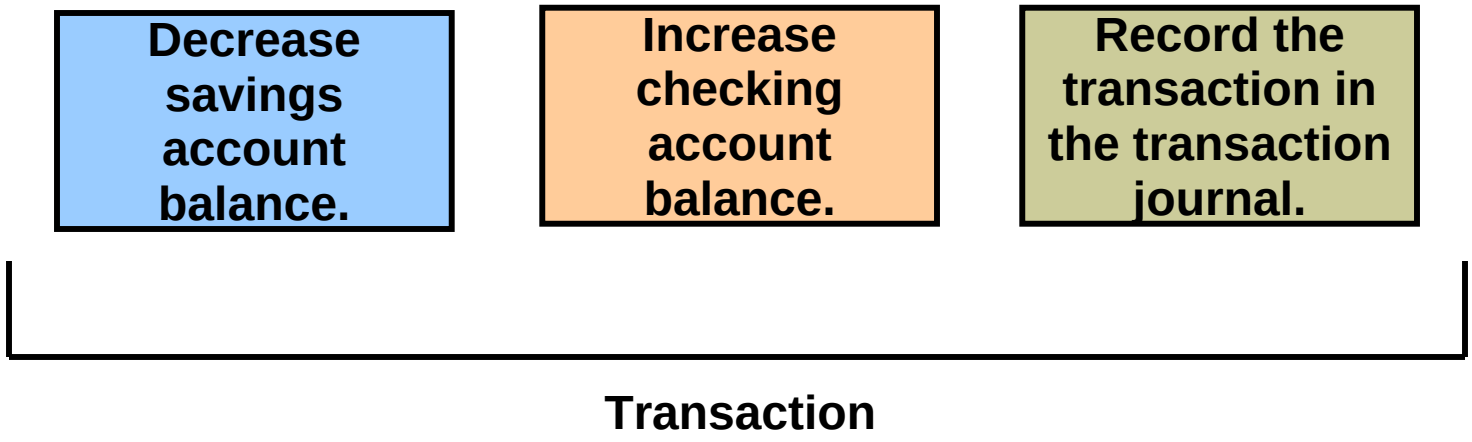
Database Transaction

A transaction is an inseparable list of database operations that must be executed either in its entirety or not at all. Transactions maintain data integrity and guarantee that the database is always in a consistent state.

Tell Me/Show Me

Example of a Transaction

To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:



Tell Me/Show Me

Example of a Transaction (continued)

What would happen if there were insufficient funds in the savings account? Would the funds still be added to the checking account? Would an entry be logged in the transaction journal? What do you think *should* happen?

Decrease savings account.

```
UPDATE savings_accounts
SET balance = balance - 500
WHERE account = 3209;
```

Increase checking account.

```
UPDATE checking_accounts
SET balance = balance + 500
WHERE account = 3208;
```

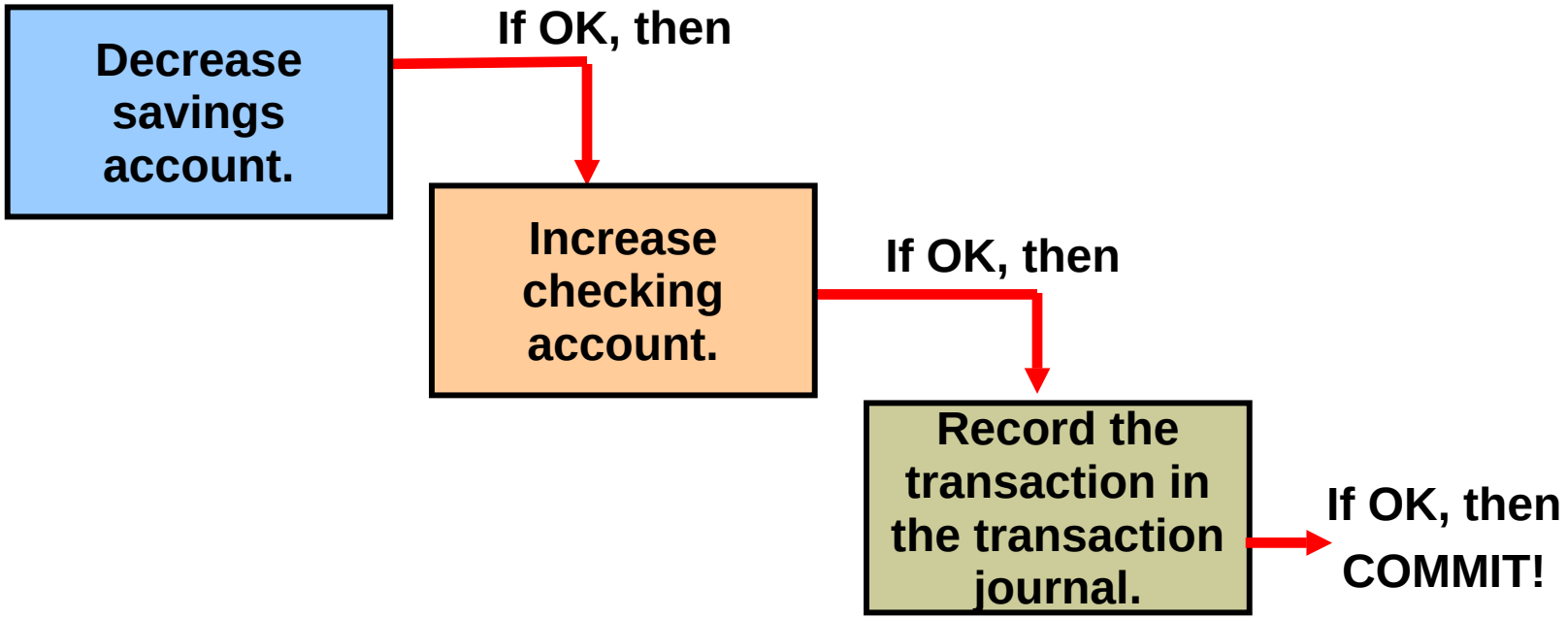
Record the transaction in the transaction journal.

```
INSERT INTO journal VALUES
(journal_seq.NEXTVAL, '1B'
3209, 3208, 500);
```

Tell Me/Show Me

Example of a Transaction (continued)

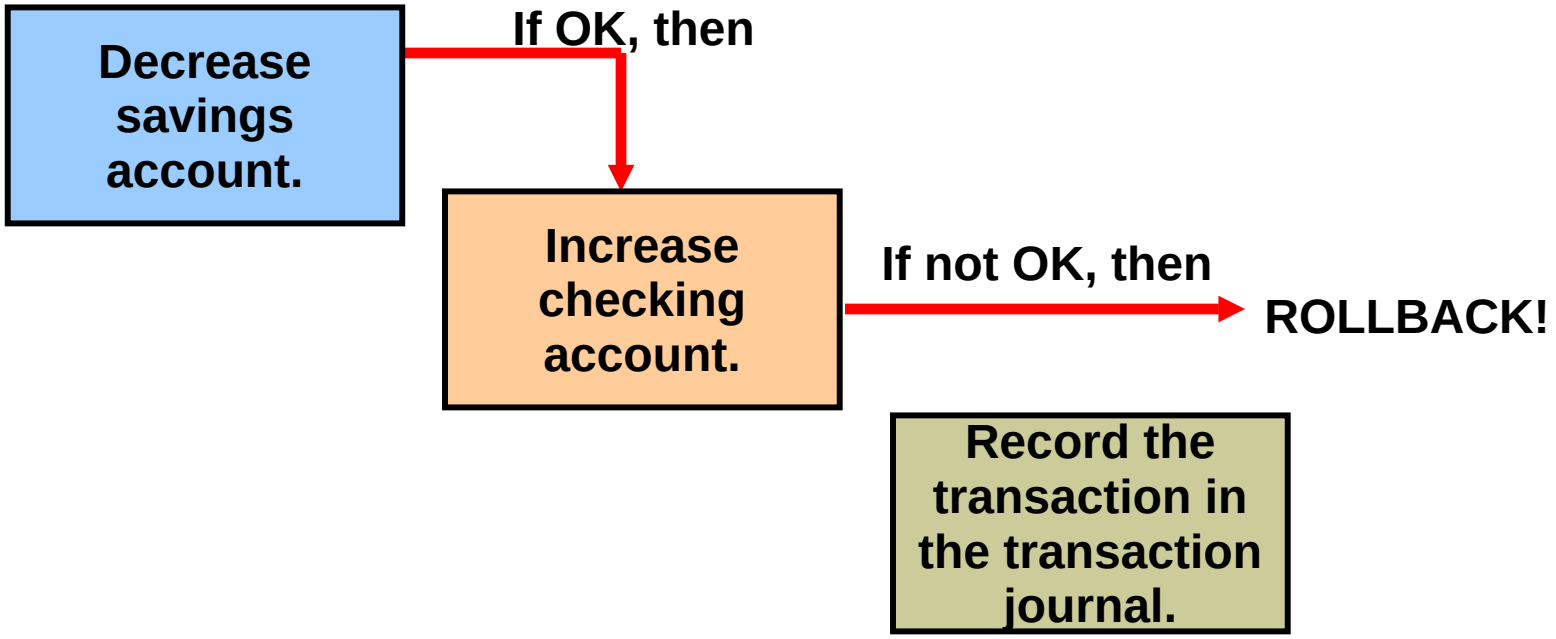
If all three SQL statements can be performed to maintain the accounts in proper balance, the effects of the transaction can be committed, or applied to the database tables.



Tell Me/Show Me

Example of a Transaction (continued)

However, if a problem, such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, the entire transaction must be rolled back (reversed out) so that the balance of all accounts is correct.



Tell Me/Show Me

Transaction Control Statements

You use transaction control statements to make the changes to the database permanent or to discard them. The three main transaction control statements are:

- `COMMIT`
- `ROLLBACK`
- `SAVEPOINT`

The transaction control commands are valid in PL/SQL and therefore can be used directly in the executable or exception section of a PL/SQL block.

Tell Me/Show Me

COMMIT

COMMIT is used to make the database changes permanent. If a transaction ends with a COMMIT statement, all the changes made to the database during that transaction are made permanent.

```
BEGIN
  INSERT INTO pairtable VALUES (1, 2);
  COMMIT;
END;
```

Note: The keyword END signals the end of a PL/SQL block, not the end of a transaction.

Tell Me/Show Me

ROLLBACK

ROLLBACK is for discarding any changes that were made to the database after the last COMMIT. If the transaction fails, or ends with a ROLLBACK, then none of the statements takes effect.

```
BEGIN
  INSERT INTO pairtable VALUES (3, 4);
  ROLLBACK;
  INSERT INTO pairtable VALUES (5, 6);
  COMMIT;
END;
```

In the example, only the second INSERT statement adds a row of data.

Tell Me/Show Me

SAVEPOINT

SAVEPOINT is used to mark an intermediate point in transaction processing.

```
BEGIN
  INSERT INTO pairtable VALUES (7, 8);
  SAVEPOINT my_sp_1;
  INSERT INTO pairtable VALUES (9, 10);
  SAVEPOINT my_sp_2;
  INSERT INTO pairtable VALUES (11, 12);
  ROLLBACK to my_sp_1;
  INSERT INTO pairtable VALUES (13, 14);
  COMMIT;
END;
```

Only ROLLBACK can be used to a SAVEPOINT.

Conditional Control: `IF` Statements

Tell Me/Show Me

IF Statements

The structure of the PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It enables PL/SQL to perform actions selectively based on conditions.

Syntax:

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Tell Me/Show Me

IF Statements (continued)

- ELSIF is a keyword that introduces a Boolean expression. (If the first condition yields FALSE or NULL, then the ELSIF keyword introduces additional conditions.)
- ELSE introduces the default clause that is executed if and only if none of the earlier conditions (introduced by IF and ELSIF) are TRUE. The tests are executed in sequence so that a later condition that might be true is pre-empted by an earlier condition that is true.
- END IF; marks the end of an IF statement.

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Tell Me/Show Me

IF ELSIF ELSE Clause

The IF statement now contains multiple ELSIF clauses and an ELSE clause. Notice that the ELSIF clauses add additional conditions. As with the IF, each ELSIF condition is followed by a THEN clause, which is executed if the condition returns TRUE.

```
DECLARE
  v_myage NUMBER:=31;
BEGIN
  IF v_myage < 11
    THEN
      DBMS_OUTPUT.PUT_LINE('I am a child');
  ELSIF v_myage < 20
    THEN
      DBMS_OUTPUT.PUT_LINE('I am young');
  ELSIF v_myage < 30
    THEN
      DBMS_OUTPUT.PUT_LINE('I am in my twenties');
  ELSIF v_myage < 40
    THEN
      DBMS_OUTPUT.PUT_LINE('I am in my thirties');
  ELSE
    DBMS_OUTPUT.PUT_LINE('I am always young ');
  END IF;
END;
```

Tell Me/Show Me

IF ELSIF ELSE Clause (continued)

When you have multiple clauses in the IF statement and a condition is FALSE or NULL, control then shifts to the next clause. Conditions are evaluated one by one from the top. IF all conditions are FALSE or NULL, then the statements in the ELSE clause are executed. The final ELSE clause is optional.

```
…IF    v_myage < 11 THEN
     DBMS_OUTPUT.PUT_LINE(' I am a child ');
 ELSIF v_myage < 20 THEN
     DBMS_OUTPUT.PUT_LINE(' I am young ');
 ELSIF v_myage < 30 THEN
     DBMS_OUTPUT.PUT_LINE(' I am in my twenties ');
 ELSIF v_myage < 40 THEN
     DBMS_OUTPUT.PUT_LINE(' I am in my thirties ');
 ELSE
     DBMS_OUTPUT.PUT_LINE(' I am always young ');
END IF;…
```

Tell Me/Show Me

NULL Values in IF Statements

In this example, the v_myage variable is declared but is not initialized. The condition in the IF statement returns NULL, and not TRUE or FALSE. In such a case, the control goes to the ELSE statement because just like FALSE, NULL is not TRUE.

```
DECLARE
  v_myage NUMBER;
BEGIN
  IF v_myage < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
```

Tell Me/Show Me

Handling Nulls

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- In conditional control statements, if a condition yields `NULL`, it behaves just like a `FALSE`, and the associated sequence of statements is not executed.

Tell Me/Show Me

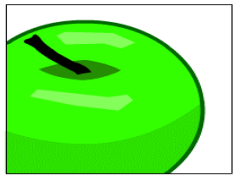
Handling Nulls

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- In conditional control statements, if a condition yields `NULL`, it behaves just like a `FALSE`, and the associated sequence of statements is not executed.

Conditional Control: CASE Statements

Tell Me/Show Me



CASE Expressions

A CASE expression selects one of a number of results and returns it into a variable.

In the syntax, *expressionN* can be a literal value, such as 50, or an expression, such as (27+23) or (v_other_var*2).

```
variable_name :=
  CASE selector
    WHEN expression1 THEN result1
    WHEN expression2 THEN result2
    ...
    WHEN expressionN THEN resultN
   [ELSE resultN+1]
  END;
```

Tell Me/Show Me

CASE Expressions: A Second Example

```
DECLARE
   v_grade     CHAR(1) := 'A';
   v_appraisal VARCHAR2(20);
BEGIN
   v_appraisal :=
      CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade ||
                         ' Appraisal ' || v_appraisal);
END;
```

```
Grade: A
Appraisal Excellent

Statement processed.
```

Tell Me/Show Me

Searched CASE Expressions

PL/SQL also provides a searched CASE expression, which has the following form:

```
CASE
   WHEN search_condition1 THEN result1
   WHEN search_condition2 THEN result2
   ...
   WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

A searched CASE expression has no selector. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type.

Tell Me/Show Me

Searched CASE Expressions: An Example

```
DECLARE
  v_grade     CHAR(1) := 'A';
  v_appraisal VARCHAR2(20);
BEGIN
  v_appraisal :=
     CASE                      -- no selector here
       WHEN v_grade = 'A' THEN 'Excellent'
       WHEN v_grade IN ('B','C') THEN 'Good'
       ELSE 'No such grade'
     END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade ||
                         ' Appraisal ' || v_appraisal);
END;
```

Tell Me/Show Me

How are CASE Expressions Different From CASE Statements?

- **CASE expressions** return a value into a variable.
- CASE expressions end with END;
- A CASE expression is a single PL/SQL statement.

```
DECLARE
  v_grade     CHAR(1) := 'A';
  v_appraisal VARCHAR2(20);
BEGIN
  v_appraisal :=
    CASE
      WHEN v_grade = 'A' THEN 'Excellent'
      WHEN v_grade IN ('B','C') THEN 'Good'
      ELSE 'No such grade'
    END;
  DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade ||
                        ' Appraisal ' || v_appraisal);
END;
```

Tell Me/Show Me

How are CASE Expressions Different From CASE Statements? (continued)

- **CASE statements** evaluate conditions and perform actions
- A CASE statement can contain many PL/SQL statements.
- CASE statements end with END CASE;.

```
DECLARE
   v_grade CHAR(1) := 'A';
BEGIN
  CASE
    WHEN v_grade = 'A' THEN
          DBMS_OUTPUT.PUT_LINE ('Excellent');
    WHEN v_grade IN ('B','C') THEN
          DBMS_OUTPUT.PUT_LINE ('Good');
    ELSE
          DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
```

Tell Me/Show Me

Logic Tables

When using IF and CASE statements you often need to combine conditions using AND, OR, and NOT. The following Logic Tables show the results of all possible combinations of two conditions.

AND	TRUE	FALSE	NULL
TRUE	TRUE	(1) FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

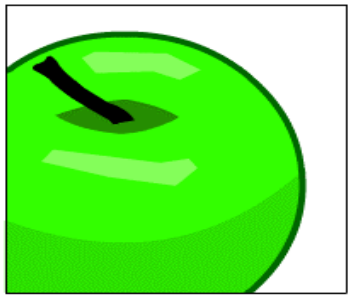
OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Example: (1) TRUE AND FALSE is FALSE

Iterative Control: Basic Loops

Tell Me/Show Me



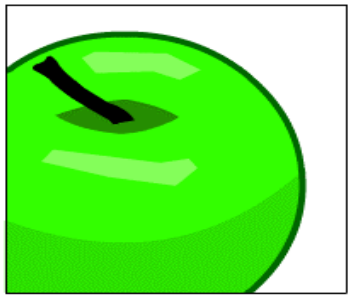
Iterative Control: LOOP Statements

Loops repeat a statement or a sequence of statements multiple times.

PL/SQL provides the following types of loops:

- Basic loops that perform repetitive actions without overall conditions
- FOR loops that perform iterative actions based on a counter
- WHILE loops that perform repetitive actions based on a condition

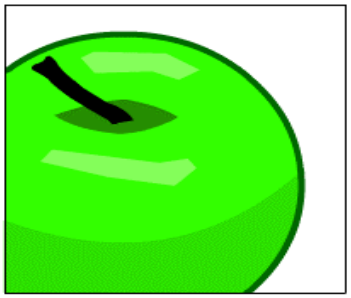
Tell Me/Show Me



Basic Loops

The simplest form of a LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP. Use the basic loop when the statements inside the loop must execute at least once.

Tell Me/Show Me



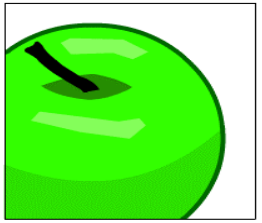
Basic Loops

Each time the flow of execution reaches the END LOOP statement, control is returned to the corresponding LOOP statement above it. A basic loop allows the execution of its statements at least once, even if the EXIT condition is already met upon entering the loop. Without the EXIT statement, the loop would be infinite.

Syntax:

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

Tell Me/Show Me



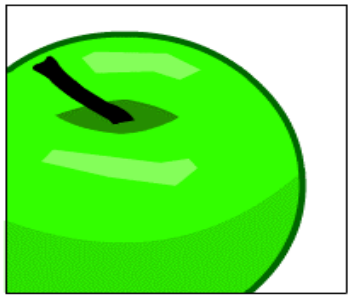
Basic Loops

The EXIT Statement

You can use the EXIT statement to terminate a loop. The control passes to the next statement after the END LOOP statement. You can issue EXIT either as an action within an IF statement or as a stand-alone statement within the loop.

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('The square of '
              ||v_counter||' is: '|| POWER(v_counter,2));
    v_counter := v_counter + 1;
    IF v_counter > 10 THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

Tell Me/Show Me

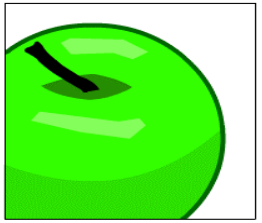


Basic Loops

The EXIT Statement (continued)

- The EXIT statement must be placed inside a loop.
- If the EXIT condition is placed at the top of the loop (before any of the other executable statements) and that condition is initially true, then the loop exits and the other statements in the loop never execute.
- A basic loop can contain multiple EXIT statements, but you should have only one EXIT point.

Tell Me/Show Me



Basic Loops

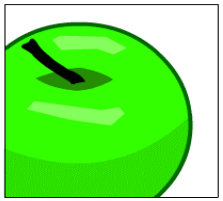
The EXIT WHEN Statement

Use the WHEN clause to allow conditional termination of the loop. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition yields TRUE, then the loop ends and control passes to the next statement after the loop.

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('The square of '
              ||v_counter||' is: '|| POWER(v_counter,2));
    v_counter :=v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

Iterative Control: `WHILE` and `FOR` Loops

Tell Me/Show Me



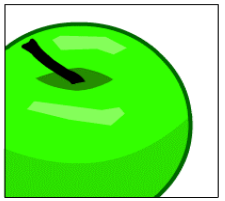
WHILE Loops:

You can use the WHILE loop to repeat a sequence of statements until the controlling condition is no longer TRUE. The condition is evaluated at the start of each iteration. The loop terminates when the condition is FALSE or NULL. If the condition is FALSE or NULL at the start of the loop, then no further iterations are performed.

Syntax:

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Tell Me/Show Me

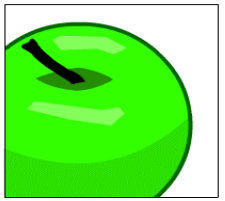


WHILE Loops (continued):

- In the syntax:
  - *condition* is a Boolean variable or expression (TRUE, FALSE, or NULL)
  - *statement* can be one or more PL/SQL or SQL statements
- If the variables involved in the conditions do not change during the body of the loop, then the condition remains TRUE and the loop does not terminate.
- **Note:** If the condition yields NULL, then the loop is bypassed and the control passes to the next statement.

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Tell Me/Show Me



FOR Loops:

FOR loops have the same general structure as the basic loop. In addition, they have a control statement before the LOOP keyword to set the number of iterations that PL/SQL performs.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.
- *lower_bound .. upper_bound* is the required syntax.

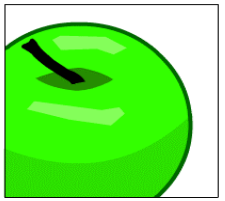
Tell Me/Show Me

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

FOR Loops (continued):

- In the syntax:
  - *Counter* is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached.
  - REVERSE causes the counter to decrement with each iteration from the upper bound to the lower bound. (Note that the lower bound is still referenced first.)
  - *lower_bound* specifies the lower bound for the range of counter values.
  - *upper_bound* specifies the upper bound for the range of counter values.
- Do not declare the counter; it is declared implicitly as an integer.

Tell Me/Show Me

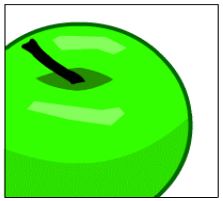


FOR Loops (continued):

- **Note:** The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower bound and upper bound of the loop range can be literals, variables, or expressions, but must evaluate to integers. The bounds are rounded to integers—that is, 11/3 or 8/5 are valid upper or lower bounds. The lower bound and upper bound are inclusive in the loop range. If the lower bound of the loop range evaluates to a larger integer than the upper bound, then the sequence of statements will not be executed.
  For example, the following statement is executed only once:

```
FOR i in 3..3
LOOP
  statement1;
END LOOP;
```

Tell Me/Show Me



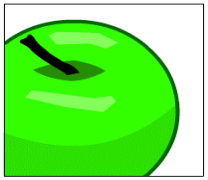
FOR Loops:

Guidelines

- Reference the counter within the loop only; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.
- Neither loop bound should be NULL.

Iterative Control: Nested Loops

Tell Me/Show Me



Nested Loops

In PL/SQL, you can nest loops to multiple levels. You can nest `FOR`, `WHILE`, and basic loops within one another.

Consider the following example:

```
BEGIN
  FOR v_outerloop IN 1..3 LOOP
    FOR v_innerloop IN REVERSE 1..5 LOOP
     DBMS_OUTPUT.PUT_LINE('Outer loop is:'||v_outerloop||
                          ' and inner loop is: '||v_innerloop);
    END LOOP;
  END LOOP;
END;
```

Tell Me/Show Me

Nested Loops

This example contains EXIT conditions in nested basic loops.

```
DECLARE
  v_outer_done   CHAR(3) := 'NO';
  v_inner_done   CHAR(3) := 'NO';
BEGIN
  LOOP                -- outer loop
    ...
    LOOP              -- inner loop
      ...
      ...             -- step A
      EXIT WHEN v_inner_done = 'YES';
      ...

    END LOOP;
    ...
    EXIT WHEN v_outer_done = 'YES';
    ...
  END LOOP;
END;
```

What if you want to exit from the outer loop at step A?

Tell Me/Show Me

Use labels to distinguish between the loops:

```
DECLARE
  ...
BEGIN
 <<outer_loop>>
  LOOP                -- outer loop
    ...
    <<inner_loop>>
    LOOP              -- inner loop
      EXIT outer_loop WHEN ...  -- Exits both loops
      EXIT WHEN v_inner_done = 'YES';
      ...
    END LOOP;
    ...
    EXIT WHEN v_outer_done = 'YES';
    ...
  END LOOP;
END;
```

Tell Me/Show Me

Loop Labels

Loop label names follow the same rules as other identifiers. A label is placed before a statement, either on the same line or on a separate line. In `FOR` or `WHILE` loops, place the label before `FOR` or `WHILE` within label delimiters (<<*label*>>). If the loop is labeled, the label name can optionally be included after the `END LOOP` statement for clarity.

Tell Me/Show Me

Labels

Label basic loops by placing the label before the word LOOP within label delimiters (<<*label*>>).

```
DECLARE
  v_outerloop PLS_INTEGER :=0;
  v_innerloop PLS_INTEGER :=5;
BEGIN
 <<Outer_loop>>
  LOOP
    v_outerloop := v_outerloop + 1;
    v_innerloop := 5;
    EXIT WHEN v_outerloop > 3;
    <<Inner_loop>>
    LOOP
      DBMS_OUTPUT.PUT_LINE('Outer loop is:'||v_outerloop||
                    ' and inner loop is: '||v_innerloop);
      v_innerloop := v_innerloop - 1;
      EXIT WHEN v_innerloop =0;
    END LOOP Inner_loop;
  END LOOP Outer_loop;
END;
```

User-Defined Records

Tell Me/Show Me

PL/SQL Records

A PL/SQL record is a composite data type consisting of a group of related data items stored as fields, each with its own name and data type. You can refer to the whole record by its name and/or to individual fields by their names.

Using %ROWTYPE implicitly declares a record whose fields match the corresponding columns by name and data type. You can reference individual fields by prefixing the field-name with the record-name:

```
... IF p_emp_record.salary > 25000 THEN
      RAISE_APPLICATION_ERROR
        (-20104, 'This employee earns too much!');
    END IF; ...
```

Tell Me/Show Me

Defining Your Own Records

But what if your example procedure SELECTs from a join of several tables?

You can declare your own record structures containing any fields you like. PL/SQL records:

- Must contain one or more components (fields) of any scalar or composite type
- Are not the same as rows in a database table
- Can be assigned initial values and can be defined as NOT NULL
- A record can be a component of another record (nested records).

Tell Me/Show Me

Creating a User-Defined PL/SQL Record

A record structure is a composite data type, just as DATE, VARCHAR2, NUMBER, and so on are Oracle-defined scalar data types. You declare the type and then declare one or more variables of that type.

```
TYPE type_name IS RECORD
     (field_declaration[,field_declaration]...);

identifier   type_name;
```

field_declaration can be of any PL/SQL data type, including %TYPE, %ROWTYPE, and RECORD.

Tell Me/Show Me

User-Defined PL/SQL Records Example

```
TYPE person_type IS RECORD
     (first_name  employees.first_name%TYPE,
      last_name   employees.last_name%TYPE,
      gender      VARCHAR2(6));
TYPE employee_type IS RECORD
     (job_id      VARCHAR2(10),
      salary      number(8,2),
      person_data person_type);

person_rec      person_type;
employee_rec    employee_type;
...
  IF person_rec.last_name ... END IF;
  employee_rec.person_data.last_name := ...;
```

Tell Me/Show Me

Where Can Types and Records Be Declared and Used?

They are composite variables and can be declared anywhere that scalar variables can be declared: in anonymous blocks, procedures, functions, package specifications (global), package bodies (local), triggers, and so on.

Their scope and visibility follows the same rules as for scalar variables. For example, you can declare a type in a package specification. Records based on that type can be declared and used anywhere within the package, and also in the calling environment.

The next two slides show an example of this.

Tell Me/Show Me

```
CREATE OR REPLACE PACKAGE pers_pack IS
  TYPE person_type IS RECORD
     (first_name  employees.first_name%TYPE,
      last_name   employees.last_name%TYPE,
      gender      VARCHAR2(6));
  PROCEDURE pers_proc (p_pers_rec OUT person_type);
END pers_pack;
CREATE OR REPLACE PACKAGE BODY pers_pack IS
  PROCEDURE pers_proc (p_pers_rec OUT person_type) IS
      v_pers_rec     person_type;
    BEGIN
      SELECT first_name, last_name, 'Female' INTO
v_pers_rec
        FROM employees WHERE employee_id = 100;
    p_pers_rec := v_pers_rec;
    END pers_proc;
END pers_pack;
```

Tell Me/Show Me

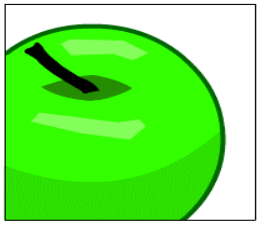
Visibility and Scope of Records Example (continued)

Now invoke the package procedure from another PL/SQL block (it could be a Java, C, or other language application):

```
DECLARE
  a_pers_rec       pers_pack.person_type;          -- 1
BEGIN
  pers_pack.pers_proc(a_pers_rec);                 -- 2
  DBMS_OUTPUT.PUT_LINE
     (a_pers_rec.first_name ||' '||a_pers_rec.gender);
END;
```

Index by Tables of Records

Tell Me/Show Me



What is a Collection?

A collection is a set of occurrences of the same kind of data. For example, the set of all employees' last names. Or, the set of all department rows.

In PL/SQL, a collection is a type of composite variable, just like user-defined records and %ROWTYPE. You see two kinds of collections in this lesson:

- An INDEX BY TABLE, which is based on a single field or column, for example on the `last_name` column of EMPLOYEES.
- An INDEX BY TABLE OF RECORDS, which is based on a composite record type, for example on the whole DEPARTMENTS row.

Because collections are PL/SQL variables, their data is stored in a private memory area like any other PL/SQL variable.

Tell Me/Show Me

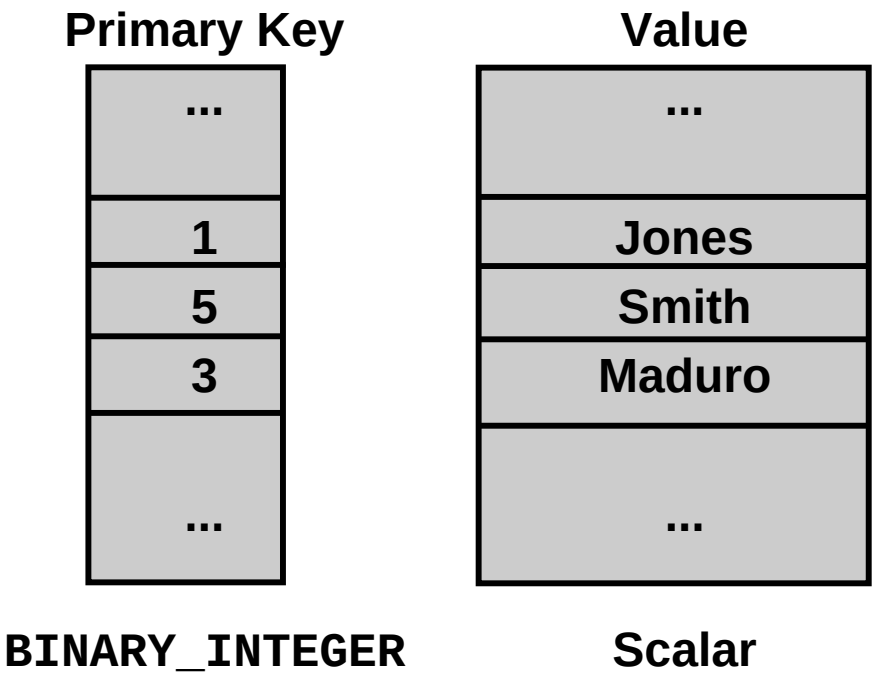
An INDEX BY Table Has a Primary Key

You need to able to distinguish between individual values in the table, so that you can reference them individually. Therefore, every `INDEX BY` table automatically has a numeric primary key, which serves as an index into the table.

The primary key must be of datatype `BINARY_INTEGER` (the default) or `PLS_INTEGER`. The primary key can be negative as well as positive.

Tell Me/Show Me

INDEX BY Table Structure



The primary key must be numeric, but could be meaningful business data, for example an employee id.

Tell Me/Show Me

Declaring an INDEX BY Table

```
DECLARE TYPE t_names IS TABLE OF VARCHAR2(50)
                        INDEX BY BINARY_INTEGER;
  last_names_tab   t_names;
  first_names_tab  t_names;
```

Like user-defined records, you must first declare a type and then declare “real” variables of that type.

This example declares two INDEX BY tables of the same type.

Tell Me/Show Me

Populating an INDEX BY Table

```
DECLARE
 TYPE t_names IS  TABLE OF VARCHAR2(50)
                  INDEX BY BINARY_INTEGER;
 last_names_tab   t_names;
BEGIN
 FOR emp_rec IN (SELECT employee_id, last_name
                   FROM employees) LOOP
   last_names_tab(emp_rec.employee_id) := emp_rec.last_name;
  END LOOP;
END;
```

This example populates the INDEX BY table with employees' last names, using `employee_id` as the primary key.

Tell Me/Show Me

Using INDEX BY Table Methods

You can use built-in procedures and functions (called methods) to reference single elements of the table, or to read successive elements. The available methods are:

- EXISTS
- COUNT
- FIRST and LAST
- PRIOR
- NEXT
- DELETE
- TRIM

You use these methods by dot-prefixing the method-name with the table-name. The next slide shows some examples.

Tell Me/Show Me

Using INDEX BY Table Methods (continued)

```
DECLARE
 TYPE t_names IS  TABLE OF VARCHAR2(50)
                  INDEX BY BINARY_INTEGER;
 last_names_tab   t_names;
 v_count          INTEGER;
BEGIN
 -- populate the INDEX BY table with employee data as before
  v_count := last_names_tab.COUNT;                          --1
  FOR i IN last_names_tab.FIRST .. last_names_tab.LAST --2
   LOOP
    IF last_names_tab.EXISTS(i) THEN                        --3
      DBMS_OUTPUT.PUT_LINE(last_names_tab(i));
    END IF;
   END LOOP;
END;
```

Tell Me/Show Me

INDEX BY TABLE OF RECORDS

Even though an index by table can have only one data field, that field can be a composite data type, such as a RECORD.

The record can be %ROWTYPE or a user-defined record.

This example declares an INDEX BY table to store complete employee rows:

```
DECLARE
  TYPE t_emprec IS TABLE OF employees%ROWTYPE
                   INDEX BY BINARY_INTEGER;
 employees_tab   t_emprec;
```

Tell Me/Show Me

Using an INDEX BY Table of Records

```
DECLARE
  TYPE t_emprec IS  TABLE OF employees%ROWTYPE
                    INDEX BY BINARY_INTEGER;
 employees_tab   t_emprec;
BEGIN
  FOR emp_rec IN (SELECT * FROM employees) LOOP
   employees_tab(emp_rec.employee_id) := emp_rec;          --1
  END LOOP;
 FOR i IN employees_tab.FIRST .. employees_tab.LAST
   LOOP
    IF employees_tab.EXISTS(i) THEN
       DBMS_OUTPUT.PUT_LINE(employees_tab(i).first_name); --2
    END IF;
   END LOOP;
END;
```