PL-SQL

PL/SQL Block Structure (Oracle 26ai)

Definition:

- **PL/SQL** → Oracle's procedural extension to SQL.
- A **PL/SQL block** is a **unit of code** that can include declarations, executable statements, and exception handling.

Think: "It's like a small program inside Oracle to **process data step by step**."

Why It's Important

- Combine **SQL + procedural logic** (loops, conditions, variables)
- Essential for **automation, stored procedures, triggers**
- Commonly asked in **fresher interviews**

Key Points

- **Anonymous block** → temporary block, not stored in DB
- **Named block** → stored as **procedure, function, or package**
- **Sections**: Declaration, Execution, Exception

PL/SQL Block Structure

```
DECLARE
   -- Declaration Section
   variable_name datatype [:= value];
BEGIN
   -- Executable Section
   -- SQL & PL/SQL statements
   statement1;
   statement2;
EXCEPTION
   -- Exception Handling Section
   WHEN exception_name1 THEN
       -- handle exception
   WHEN exception_name2 THEN
       -- handle exception
END;
```

Sections Explained

Section	Purpose	Example
Declaration	Declare variables, constants, cursors	`v_salary NUMBER := 50000;`
Execution	Main logic of program	`` `DBMS_OUTPUT.PUT_LINE('Salary: ' ``
Exception	Handle runtime errors	`WHEN ZERO_DIVIDE THEN DBMS_OUTPUT.PUT_LINE('Cannot divide by zero');`

Example: Simple PL/SQL Block

```
DECLARE
   v_name VARCHAR2(50) := 'Ali';
   v_salary NUMBER := 50000;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Employee: ' || v_name);
   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
EXCEPTION
```

```
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred.');
END;
```

- Declares variables
- Prints values using DBMS_OUTPUT.PUT_LINE
- Handles **any runtime error**

Real-Life Usage

- HR → Calculate bonuses for all employees
- Sales → Update sales commission
- Education → Generate student marks reports
- Analytics → Automate daily report generation

Interview Questions

Q1. What are the three sections of a PL/SQL block?

- DECLARE, BEGIN (Execution), EXCEPTION

Q2. Is DECLARE mandatory?

- No, only BEGIN-EXCEPTION section is required

Q3. Difference between anonymous and named block?

- Anonymous → temporary, not stored
- Named → stored in DB, reusable (procedures, functions)

Q4. How to handle errors in PL/SQL?

- Use **EXCEPTION section** with WHEN exception_name THEN

Common Fresher Mistakes

- Forgetting **END;** → syntax error
- Using **uninitialized variables**
- Ignoring exceptions → runtime errors
- Using SQL statements incorrectly inside BEGIN

Best Practices

- Always use **EXCEPTION handling**
- Initialize variables in DECLARE section
- Use **DBMS_OUTPUT.PUT_LINE** for debugging
- Keep block modular → smaller blocks for readability
- Use meaningful **variable names**

PL/SQL %TYPE & %ROWTYPE (Oracle 26ai)

Definition:

Feature	Definition
%TYPE	Allows a PL/SQL variable to inherit the **data type of a column** in a table or another variable.
%ROWTYPE	Declares a variable to hold **an entire row** of a table or cursor.

Think:

- %TYPE → “One column’s type”
- %ROWTYPE → “One row’s structure”

Why It’s Important

- Ensures **data type consistency** with table columns
- Avoids **hardcoding data types** → easier maintenance

- Essential for **PL/SQL programming** in real applications
- Frequently asked in **fresher interviews**

Key Points

Feature	Key Points
%TYPE	- Variable inherits column type- Reduces errors if column type changes- Can refer to table columns or another variable
%ROWTYPE	- Holds all columns of a table or cursor- Useful for fetching entire row in one variable- Can be used with cursors and tables

---

Syntax & Examples

%TYPE Example

```
DECLARE
   v_employee_id employee.employee_id%TYPE; -- same type as employee_id column
   v_salary employee.salary%TYPE;           -- same type as salary column
BEGIN
   v_employee_id := 101;
   v_salary := 50000;
   DBMS_OUTPUT.PUT_LINE('ID: ' || v_employee_id || ', Salary: ' || v_salary);
END;
```

%ROWTYPE Example

```
DECLARE
   v_emp employee%ROWTYPE;  -- variable to hold an entire employee row
BEGIN
   SELECT * INTO v_emp
   FROM employee
   WHERE employee_id = 101;

   DBMS_OUTPUT.PUT_LINE('Name: ' || v_emp.name || ', Salary: ' || v_emp.salary);
END;
```

- Fetches **all columns** of a single employee row
- Easier than declaring separate variables for each column

Real-Life Usage

- HR → Fetch entire employee record for processing
- Sales → Process order row-by-row using cursors
- Reporting → Use %ROWTYPE for batch updates
- Avoids data type mismatch when table schema changes

Interview Questions

Q1. Difference between %TYPE and %ROWTYPE?

- %TYPE → single column data type
- %ROWTYPE → entire row structure

Q2. Can %ROWTYPE be used with cursors?

- Yes, cursor_name%ROWTYPE can hold current row of a cursor

Q3. Why use %TYPE/%ROWTYPE instead of explicit data types?

- Avoids errors if table column type changes
- Easier to maintain and read

Common Fresher Mistakes

- Forgetting table/column name in %TYPE → syntax error
- Using %ROWTYPE for multiple rows → only works for **single row fetch**
- Not using SELECT INTO with %ROWTYPE → runtime error
- Confusing %TYPE and %ROWTYPE

Best Practices

- Always use **%TYPE** for single column variables
- Use **%ROWTYPE** to fetch **entire row**
- Combine with cursors for row-wise processing
- Avoid hardcoding column data types → reduces maintenance issues
- Use meaningful variable names for readability

PL/SQL Control Structures (Oracle 26ai)

Definition:

- **Control structures** in PL/SQL are used to **control the flow of execution.**
- Includes **conditional statements** and **loops.**

Think: "Decide what to do next based on conditions or repeat actions until done."

Why It's Important

- Makes PL/SQL **dynamic and decision-based**
- Essential for **automation, validation, and batch processing**
- Core for **fresher interviews**

Key Points

- **Conditional statements** → decide which code to execute
- **Loops** → repeat code multiple times
- **EXIT** → terminate loops
- Can be combined with **cursors, variables, exceptions**

Conditional Statements

IF Statement

```
IF condition THEN
   -- statements
ELSIF another_condition THEN
   -- statements
ELSE
   -- statements
END IF;
```

Example

```
DECLARE
   v_salary NUMBER := 50000;
BEGIN
   IF v_salary > 60000 THEN
      DBMS_OUTPUT.PUT_LINE('High Salary');
   ELSIF v_salary BETWEEN 40000 AND 60000 THEN
      DBMS_OUTPUT.PUT_LINE('Medium Salary');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Low Salary');
   END IF;
END;
```

CASE Statement

```
CASE expression
   WHEN value1 THEN statement1;
   WHEN value2 THEN statement2;
```

```
   ELSE statement_default;
END CASE;
```

Example

```
DECLARE
   v_grade CHAR := 'B';
BEGIN
   CASE v_grade
      WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
      WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Good');
      ELSE DBMS_OUTPUT.PUT_LINE('Average');
   END CASE;
END;
```

Loops in PL/SQL

Loop Type	Description
Basic LOOP	Executes statements until EXIT condition is met
WHILE LOOP	Executes while condition is TRUE
FOR LOOP	Executes for a specified number of iterations

Basic LOOP

```
DECLARE
   v_counter NUMBER := 1;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
      v_counter := v_counter + 1;
      EXIT WHEN v_counter > 5;
   END LOOP;
END;
```

WHILE LOOP

```
DECLARE
   v_counter NUMBER := 1;
BEGIN
   WHILE v_counter <= 5 LOOP
      DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
      v_counter := v_counter + 1;
   END LOOP;
END;
```

FOR LOOP

```
BEGIN
   FOR v_counter IN 1..5 LOOP
      DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
   END LOOP;
END;
```

Real-Life Usage

- HR → Process salaries, classify employees by grade
- Sales → Loop through orders, calculate totals
- Education → Loop through students, assign grades
- Automation → Execute repetitive tasks with loops

Interview Questions

Q1. Difference between IF and CASE?

- IF → good for **complex logical conditions**
- CASE → better for **single expression with multiple values**

Q2. Difference between FOR and WHILE loops?

- FOR → fixed number of iterations
- WHILE → loop until condition becomes false

Q3. Can loops be nested?

- Yes, loops can be **nested inside another loop**

Q4. How to exit a loop?

- Use **EXIT WHEN condition**

Common Fresher Mistakes

- Forgetting **END IF;** or **END LOOP;**
- Infinite loops → missing EXIT condition
- Misusing logical conditions in IF/WHILE
- Not initializing loop variables

Best Practices

- Use **EXIT WHEN** in loops for clarity
- Use **FOR loops** when iteration count is known
- Prefer CASE for **simple value comparisons**
- Keep **nested loops simple** to avoid complexity
- Always **test conditions** carefully to prevent infinite loops

---

PL/SQL Conditional Statements: IF-THEN-ELSE & CASE (Oracle 26ai)

Definition:

- **IF-THEN-ELSE** → Executes code based on **logical conditions.**
- **CASE** → Executes code based on **specific values of an expression.**

Think:

- IF → “Complex decisions based on conditions”
- CASE → “Choose one path based on a known value”

Why It’s Important

- Makes PL/SQL **decision-driven**
- Essential for **validation, classification, and reporting**
- Frequently asked in **fresher interviews**

Key Points

Feature	IF-THEN-ELSE	CASE
Type	Conditional / Boolean	Value-based
Use	Complex logical conditions	Single expression, multiple values
Syntax	IF condition THEN … ELSIF … ELSE … END IF;	CASE expression WHEN value1 THEN … ELSE … END CASE;
Readability	Can become complex if nested	Cleaner for single-value comparisons

---

Syntax & Examples

IF-THEN-ELSE

DECLARE

```
   v_salary NUMBER := 50000;
BEGIN
   IF v_salary > 60000 THEN
      DBMS_OUTPUT.PUT_LINE('High Salary');
   ELSIF v_salary BETWEEN 40000 AND 60000 THEN
      DBMS_OUTPUT.PUT_LINE('Medium Salary');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Low Salary');
   END IF;
END;
```

- Executes based on **logical conditions**
- Can have **multiple ELSIF branches**

CASE Statement

```
DECLARE
   v_grade CHAR := 'B';
BEGIN
   CASE v_grade
      WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
      WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Good');
      WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Average');
      ELSE DBMS_OUTPUT.PUT_LINE('Below Average');
   END CASE;
END;
```

- Executes based on **specific values of v_grade**
- **ELSE** is optional but recommended

CASE as Expression in SELECT

```
SELECT employee_id, name,
       CASE
           WHEN salary > 60000 THEN 'High'
           WHEN salary BETWEEN 40000 AND 60000 THEN 'Medium'
           ELSE 'Low'
       END AS salary_grade
FROM employee;
```

- Can classify data **directly in SQL query**

Real-Life Usage

- HR → Classify employees based on salary grade
- Sales → Categorize orders as High/Medium/Low value
- Education → Assign grades to students
- Analytics → Conditional calculations in reports

Interview Questions

Q1. Difference between IF-THEN-ELSE and CASE?

- IF → good for **complex logical conditions**
- CASE → cleaner for **value comparisons**

Q2. Can CASE be used in SQL query?

- Yes, directly in **SELECT, WHERE, ORDER BY**

Q3. Can IF be used in SQL query?

- No, IF is **PL/SQL only**

Q4. Can CASE have multiple WHEN conditions for same outcome?

- Yes, by combining using **OR** in searched CASE or repeating WHEN for each value

---

Common Fresher Mistakes

- Forgetting **END IF;** or **END CASE;**
- Using IF in SQL → not allowed
- Misplacing THEN or ELSE
- Complex nested IFs → hard to read

Best Practices

- Use **CASE** for simple value-based decisions
- Use **IF-THEN-ELSE** for logical comparisons
- Avoid deep nesting → split into smaller blocks
- Always include **ELSE** to handle unexpected cases
- Test all branches for correctness

PL/SQL Loops: LOOP, WHILE, FOR (Oracle 26ai)

Definition:

- **Loops** in PL/SQL allow **repeating a set of statements** until a condition is met.
- Types: **LOOP (basic), WHILE loop, FOR loop**

Think: "Do this action multiple times until a condition is satisfied."

Why It's Important

- Automates **repetitive tasks**
- Essential for **processing multiple rows or calculations**
- Widely asked in **fresher interviews**

Key Points

Loop Type	Description	When to Use
Basic LOOP	Executes repeatedly until **EXIT** condition	Unknown number of iterations
WHILE LOOP	Executes while condition is TRUE	Condition-based iteration
FOR LOOP	Executes **fixed number of times**	Known number of iterations

- **EXIT WHEN** → stops a loop when condition is true
- Loops can be **nested**
- Can be combined with **cursors for row-wise processing**

Syntax & Examples

Basic LOOP

```
DECLARE
   v_counter NUMBER := 1;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
      v_counter := v_counter + 1;
      EXIT WHEN v_counter > 5;
   END LOOP;
END;
```

- Executes **until EXIT condition**
- Infinite if **EXIT** is missing

WHILE LOOP

```
DECLARE
   v_counter NUMBER := 1;
```

```
BEGIN
   WHILE v_counter <= 5 LOOP
      DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
      v_counter := v_counter + 1;
   END LOOP;
END;
```

- Condition is checked **before each iteration**
- Stops when **condition becomes FALSE**

FOR LOOP

```
BEGIN
   FOR v_counter IN 1..5 LOOP
      DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
   END LOOP;
END;
```

- Executes **fixed iterations** (1 to 5)
- No manual increment needed

Interview Questions

Q1. Difference between LOOP, WHILE, FOR?

- LOOP → basic, must use EXIT WHEN
- WHILE → condition checked before execution
- FOR → fixed iterations, automatically increments

Q2. Can loops be nested?

- Yes, inner loop executes for each iteration of outer loop

Q3. What happens if EXIT is missing in LOOP?

- Infinite loop → program keeps running

Q4. Can FOR loop decrement?

- Yes, use REVERSE keyword:

```
FOR i IN REVERSE 5..1 LOOP
   DBMS_OUTPUT.PUT_LINE(i);
END LOOP;
```

Common Fresher Mistakes

- Missing **EXIT WHEN** → infinite loop
- Forgetting **END LOOP;**
- Misplacing loop counter initialization
- Misusing WHILE with incorrect condition → infinite or skipped execution

---

Best Practices

- Always include **EXIT WHEN** in basic LOOP
- Prefer **FOR loop** for known iterations
- Keep **nested loops simple**
- Test loop logic separately before integrating
- Use meaningful **counter variable names**

Creating Procedures in PL/SQL (Oracle 26ai)

Definition:

- **Procedure** → A **named PL/SQL block** stored in the database that performs a **specific task.**
- Can accept **parameters** (IN, OUT, IN OUT) and can be **reused** multiple times.

Think: “A mini program in Oracle that does a task when called.”

Why It’s Important

- Reusable code → avoids repeating same logic
- Automates **business tasks** like salary calculation, order processing
- Core for **fresher interviews** on PL/SQL

Key Points

- Stored in DB → persists until explicitly dropped
- Can have **parameters**:
  - **IN** → pass value to procedure
  - **OUT** → return value from procedure
  - **IN OUT** → pass and return value
- Can include **loops, conditions, DML, cursors**
- Execute with **EXECUTE procedure_name**

Syntax

Procedure without Parameters

```
CREATE OR REPLACE PROCEDURE procedure_name IS
BEGIN
   -- SQL or PL/SQL statements
END procedure_name;
```

Procedure with Parameters

```
CREATE OR REPLACE PROCEDURE procedure_name (
   param1 IN NUMBER,
   param2 OUT VARCHAR2
) IS
BEGIN
   -- Use param1
   param2 := 'Processed value ' || param1;
END procedure_name;
```

Examples

Simple Procedure

```
CREATE OR REPLACE PROCEDURE greet_employee IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Welcome to the company!');
END greet_employee;
```

Execute

```
EXECUTE greet_employee;
```

Procedure with IN Parameter

```
CREATE OR REPLACE PROCEDURE show_salary (
   p_employee_id IN NUMBER
) IS
   v_salary employee.salary%TYPE;
BEGIN
   SELECT salary INTO v_salary
   FROM employee
   WHERE employee_id = p_employee_id;

   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
END show_salary;
```

Execute

```
EXECUTE show_salary(101);
```

Procedure with OUT Parameter

```
CREATE OR REPLACE PROCEDURE get_employee_name (
   p_employee_id IN NUMBER,
   p_name OUT VARCHAR2
) IS
BEGIN
   SELECT name INTO p_name
   FROM employee
   WHERE employee_id = p_employee_id;
END get_employee_name;
```

Execute in PL/SQL Block

```
DECLARE
   v_name VARCHAR2(50);
BEGIN
   get_employee_name(101, v_name);
   DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_name);
END;
```

Real-Life Usage

- HR → Fetch employee details
- Sales → Calculate commission for each order
- Education → Generate report cards for students
- Automation → Daily batch tasks like updating records

Interview Questions

Q1. Difference between Procedure and Function?

- Procedure → performs task, may or may not return value
- Function → always returns a value

Q2. What are procedure parameters?

- IN → pass value to procedure
- OUT → return value from procedure
- IN OUT → both pass and return

Q3. How to execute a stored procedure?

- EXECUTE procedure_name; or inside **PL/SQL block**

Q4. Can a procedure call another procedure?

- Yes, procedures can be **nested or called sequentially**

Common Fresher Mistakes

- Forgetting **IS/AS** keyword → syntax error
- Using **IN parameters as OUT** → cannot assign values
- Not committing DML changes inside procedure
- Using DBMS_OUTPUT.PUT_LINE without enabling output in SQL Developer

---

Best Practices

- Always use **CREATE OR REPLACE** for updates
- Use **meaningful procedure and parameter names**
- Handle **exceptions** inside procedure
- Commit transactions only if required
- Keep procedures **single-task oriented** → improves reusability

---

PL/SQL Procedure Parameters: IN, OUT, IN OUT (Oracle 26ai)

Definition:

- **Parameters** → Variables passed to a **procedure or function** to provide input or receive output.
- Types in PL/SQL: **IN, OUT, IN OUT**

Think: "Parameters are the way to give input to and get output from procedures/functions."

Why It's Important

- Makes procedures **dynamic and reusable**
- Pass **values in, get results out**
- Core concept for **PL/SQL interview questions**

Key Points

Parameter	Purpose	Behavior
IN	Passes value **into procedure**	Cannot be modified inside procedure
OUT	Returns value **from procedure**	Must be assigned inside procedure
IN OUT	Passes value **in** and returns **updated value**	Can be read and modified inside procedure

- Default parameter type is **IN**
- **Multiple parameters** can be used in a procedure
- Always use **meaningful names** for clarity

Syntax & Examples

IN Parameter

```
CREATE OR REPLACE PROCEDURE greet_employee (
   p_name IN VARCHAR2
) IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Welcome ' || p_name);
END greet_employee;
```

Execute

```
EXECUTE greet_employee('Ali');
```

- Input only → cannot change p_name inside procedure

OUT Parameter

```
CREATE OR REPLACE PROCEDURE get_salary (
   p_employee_id IN NUMBER,
   p_salary OUT NUMBER
) IS
BEGIN
   SELECT salary INTO p_salary
   FROM employee
   WHERE employee_id = p_employee_id;
END get_salary;
```

Execute

```
DECLARE
   v_salary NUMBER;
BEGIN
   get_salary(101, v_salary);
   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
END;
```

- Output only → returns value via p_salary

IN OUT Parameter

```
CREATE OR REPLACE PROCEDURE adjust_salary (
   p_employee_id IN NUMBER,
   p_salary IN OUT NUMBER
) IS
BEGIN
   p_salary := p_salary + 5000;  -- increment salary
   UPDATE employee
   SET salary = p_salary
   WHERE employee_id = p_employee_id;
END adjust_salary;
```

Execute

```
DECLARE
   v_salary NUMBER := 50000;
BEGIN
   adjust_salary(101, v_salary);
   DBMS_OUTPUT.PUT_LINE('Updated Salary: ' || v_salary);
END;
```

- Input value provided → updated inside procedure → output returned

---

Real-Life Usage

- HR → IN: Employee ID, OUT: Salary, IN OUT: Salary adjustment
- Sales → IN: Order ID, OUT: Total Amount
- Education → IN: Student ID, OUT: Grade, IN OUT: Marks after bonus
- Automation → Pass values, get calculated results back

Interview Questions

Q1. Difference between IN, OUT, IN OUT?

- IN → input only
- OUT → output only
- IN OUT → input + output

Q2. Can IN parameter be updated inside procedure?

- No, IN parameters are read-only

Q3. Can multiple parameters be used together?

- Yes, any combination of IN, OUT, IN OUT

Q4. Default parameter type?

- **IN**

Common Fresher Mistakes

- Trying to modify **IN parameter** → error
- Not assigning value to **OUT parameter** → returns NULL
- Forgetting to declare variable in calling block for OUT/IN OUT
- Confusing IN OUT with OUT → must initialize IN OUT

Best Practices

- Use **IN** for input values only
- Use **OUT** for returning single value
- Use **IN OUT** for update operations
- Always **initialize IN OUT variables** in calling block
- Keep procedure parameters **minimal and meaningful**

Creating Functions in PL/SQL (Oracle 26ai)

Definition:

- **Function** → A **named PL/SQL block** stored in the database that **performs a specific task and returns a value.**
- Unlike procedures, functions **must return a value** using the RETURN statement.

Think: "A mini program in Oracle that gives you a result when called."

Why It's Important

- Encapsulates **logic for reuse**
- Returns values that can be used in **SQL queries, calculations, or reports**
- Essential for **fresher interviews** on PL/SQL

Key Points

- Stored in the database → persists until dropped
- Can have **parameters**: IN, OUT (rarely), IN OUT
- Can include **loops, conditions, DML, cursors**
- Return value is **mandatory** using RETURN
- Can be used **inside SQL statements** (if deterministic and doesn't modify tables)

Syntax

```
CREATE OR REPLACE FUNCTION function_name (
   parameter1 IN datatype,
   parameter2 IN datatype
) RETURN return_datatype
IS
   -- Declaration section
   v_variable datatype;
BEGIN
   -- Logic / computation
   RETURN value;  -- must return a value
END function_name;
```

Examples

Simple Function

```
CREATE OR REPLACE FUNCTION get_bonus (
   p_salary IN NUMBER
) RETURN NUMBER
IS
BEGIN
   RETURN p_salary * 0.10;  -- 10% bonus
END get_bonus;
```

Execute in PL/SQL Block

```
DECLARE
   v_bonus NUMBER;
BEGIN
   v_bonus := get_bonus(50000);
   DBMS_OUTPUT.PUT_LINE('Bonus: ' || v_bonus);
END;
```

Function with IN Parameters

```
CREATE OR REPLACE FUNCTION employee_status (
   p_salary IN NUMBER
) RETURN VARCHAR2
IS
```

```
BEGIN
   IF p_salary > 60000 THEN
      RETURN 'High Salary';
   ELSE
      RETURN 'Moderate Salary';
   END IF;
END employee_status;
```

Usage in SQL

```
SELECT name, employee_status(salary) AS status
FROM employee;
```

- Can classify employees **directly in query**

Function with Multiple Parameters

```
CREATE OR REPLACE FUNCTION total_salary (
   p_basic IN NUMBER,
   p_bonus IN NUMBER
) RETURN NUMBER
IS
BEGIN
   RETURN p_basic + p_bonus;
END total_salary;
```

Real-Life Usage

- HR → Calculate bonuses, total salary, or tax
- Sales → Calculate total order amount or discount
- Education → Compute final grade based on marks
- Analytics → Perform calculations reusable across reports

Interview Questions

Q1. Difference between Function and Procedure?

- Function → **must return value**, can be used in SQL
- Procedure → may or may not return value

Q2. Can a function have OUT parameters?

- Rarely; usually return value via RETURN

Q3. Can a function be used in a SQL SELECT statement?

- Yes, if it **doesn't modify tables** (deterministic)

Q4. Can a function call another function or procedure?

- Yes, functions can call **other functions or procedures**

Common Fresher Mistakes

- Forgetting RETURN → compilation error
- Trying to perform DML in a function used in SQL → error
- Not specifying **RETURN datatype**
- Confusing procedure and function syntax

Best Practices

- Always use **CREATE OR REPLACE**
- Keep functions **single-task oriented**
- Use **IN parameters** for input
- Avoid DML inside functions used in SQL
- Use **meaningful function names**

PL/SQL RETURN Statement (Oracle 26ai)

Definition:

- **RETURN** → A PL/SQL statement used to **send a value back** from a **function** to the calling block or SQL statement.
- In procedures, **RETURN** can also be used to **exit the block early**.

Think: "Return is the way a function **gives its result** or a procedure **stops execution**."

Why It's Important

- Functions **must return a value** → core PL/SQL rule
- Controls **flow of execution** in procedures and functions
- Common topic in **fresher PL/SQL interviews**

Key Points

Usage	Description
In Function	Must return a value compatible with **RETURN datatype**
In Procedure	Optional → can exit the procedure early
Placement	Usually at the **end** of function or at **conditional exit** points
Behavior	Stops execution of block immediately after RETURN

Syntax & Examples

RETURN in Function

```
CREATE OR REPLACE FUNCTION calculate_bonus(
   p_salary IN NUMBER
) RETURN NUMBER
IS
BEGIN
   RETURN p_salary * 0.10;  -- 10% bonus
END calculate_bonus;
```

Execute

```
DECLARE
   v_bonus NUMBER;
BEGIN
   v_bonus := calculate_bonus(50000);
   DBMS_OUTPUT.PUT_LINE('Bonus: ' || v_bonus);
END;
```

RETURN in Procedure (exit early)

```
CREATE OR REPLACE PROCEDURE check_salary(p_salary IN NUMBER) IS
BEGIN
   IF p_salary < 10000 THEN
      DBMS_OUTPUT.PUT_LINE('Salary too low!');
      RETURN;  -- exit procedure immediately
   END IF;

   DBMS_OUTPUT.PUT_LINE('Salary is acceptable.');
END check_salary;
```

Execute

```
EXECUTE check_salary(8000);
-- Output: Salary too low!
```

RETURN with Conditional Logic in Function

```
CREATE OR REPLACE FUNCTION employee_status(
   p_salary IN NUMBER
```

```
) RETURN VARCHAR2
IS
BEGIN
   IF p_salary > 60000 THEN
      RETURN 'High Salary';
   ELSIF p_salary BETWEEN 40000 AND 60000 THEN
      RETURN 'Medium Salary';
   ELSE
      RETURN 'Low Salary';
   END IF;
END employee_status;
```

Real-Life Usage

- HR → Return bonus, tax, or salary status
- Sales → Return total order amount or discount category
- Education → Return grade based on marks
- Automation → Return computed values to calling PL/SQL blocks

---

Interview Questions

Q1. Difference between RETURN in function vs procedure?

- Function → returns value (mandatory)
- Procedure → used only to exit block (optional)

Q2. Can a function have multiple RETURN statements?

- Yes, for **conditional logic**

Q3. What happens if a function doesn't return a value?

- Compilation error → must include RETURN compatible with declared datatype

Q4. Can RETURN be used to exit loops?

- No, use **EXIT or EXIT WHEN** for loops; RETURN exits **function or procedure**

Common Fresher Mistakes

- Forgetting RETURN in function → compilation error
- Returning value of **wrong datatype**
- Using RETURN instead of EXIT in loops → logic error
- Placing RETURN outside BEGIN…END → syntax error

Best Practices

- Always **match RETURN value with function datatype**
- Use RETURN to **exit early for invalid conditions**
- Keep **conditional returns simple and clear**
- Avoid unnecessary RETURN in procedures unless needed
- Test all RETURN paths in function

PL/SQL Packages: Specification & Body (Oracle 26ai)

Definition:

- **Package** → A **group of related PL/SQL objects** (procedures, functions, variables, cursors, exceptions) stored together in the database.
- Two parts:
    1. **Package Specification (Spec)** → **Interface** → declares public objects
    2. **Package Body** → **Implementation** → defines logic of procedures/functions

Think: "Package Spec is the menu, Package Body is the kitchen that prepares the food."

Why It's Important

- Encapsulates **related logic** → improves modularity
- **Reusability** → call procedures/functions across programs
- **Security** → Spec hides internal implementation
- Core topic for **PL/SQL interviews**

Key Points

Feature	Package Specification	Package Body
Purpose	Declare public objects	Implement procedures/functions
Accessibility	Visible to users	Hidden; accessible via spec
Objects	Variables, constants, cursors, exceptions, procedures, functions	Only defines the logic for objects declared in spec
Mandatory	Yes	Optional if no procedures/functions to implement

---

Syntax & Examples

Package Specification

```
CREATE OR REPLACE PACKAGE emp_pkg IS
   -- Public variables
   v_company_name VARCHAR2(50);

   -- Procedure declaration
   PROCEDURE greet_employee(p_name IN VARCHAR2);

   -- Function declaration
   FUNCTION calculate_bonus(p_salary IN NUMBER) RETURN NUMBER;
END emp_pkg;
```

- Declares **variables, procedures, and functions**
- No implementation inside spec

Package Body

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS

   -- Initialize variable
   v_company_name VARCHAR2(50) := 'ABC Pvt Ltd';

   -- Procedure implementation
   PROCEDURE greet_employee(p_name IN VARCHAR2) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('Welcome ' || p_name || ' to ' || v_company_name);
   END greet_employee;

   -- Function implementation
   FUNCTION calculate_bonus(p_salary IN NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN p_salary * 0.10;  -- 10% bonus
   END calculate_bonus;

END emp_pkg;
```

- Implements **logic for declared procedures and functions**
- Can also define **private objects** not visible outside

Real-Life Usage

- HR → Employee management (bonus, salary, greeting)

- Sales → Order management (calculate totals, discounts)
- Education → Student reports (grades, attendance)
- Analytics → Centralized computation logic

Interview Questions

Q1. Difference between Package Spec and Package Body?

- Spec → interface, visible, declares objects
- Body → implementation, hidden, defines logic

Q2. Can you have a package without a body?

- Yes, if it contains **only public constants, variables, types** (no procedures/functions)

Q3. Can you call a procedure from a package without body?

- No, body is required for procedure/function logic

Q4. Why use packages?

- Modularity, Reusability, Security, Easier maintenance

Common Fresher Mistakes

- Forgetting to create **package body after spec** → runtime error
- Declaring procedure/function in body not in spec → error
- Confusing **public vs private objects**
- Trying to access **private body objects** outside package → error

---

Best Practices

- Always define **package spec first**
- Keep **public interface minimal** → hide unnecessary details
- Use **private objects in body** for internal logic
- Group related procedures/functions logically
- Use packages for **frequently reused operations**

PL/SQL Triggers: Timing & Events (Oracle 26ai)

Definition:

- **Trigger** → A **PL/SQL block** automatically executed when a **specific event occurs** on a table or view.
- Two main aspects: **Timing** and **Event**

Think: “Triggers are like alarms that automatically go off when something happens in the database.”

Why It’s Important

- Automates tasks → validation, auditing, logging
- Enforces **business rules** automatically
- Frequently asked in **PL/SQL and Cognizant interviews**

Key Points

Aspect	Description
Timing	When trigger executes relative to DML: BEFORE, AFTER, INSTEAD OF
Event	DML action that fires the trigger: INSERT, UPDATE, DELETE

Trigger Timing

Timing	Description	Use Case
BEFORE	Executes **before DML**	Validate data before insert/update
AFTER	Executes **after DML**	Log changes or audit table after insert/update/delete
INSTEAD OF	Replaces DML operation on **views**	Update complex views that cannot be modified directly

---

Trigger Events

Event	Description	Example
INSERT	Fires when a new row is added	Auto-generate employee ID or log new entry
UPDATE	Fires when a row is modified	Track salary changes
DELETE	Fires when a row is removed	Keep audit trail of deleted records
COMBINED (INSERT/UPDATE/DELETE)	Fires for multiple DML actions	Maintain a general audit table

---

Syntax Examples

BEFORE INSERT Trigger

```
CREATE OR REPLACE TRIGGER trg_before_insert_emp
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
   IF :NEW.salary < 10000 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Salary must be >= 10000');
   END IF;
END;
```

- **:NEW** → refers to new row values
- Validates salary **before insertion**

AFTER UPDATE Trigger

```
CREATE OR REPLACE TRIGGER trg_after_update_salary
AFTER UPDATE OF salary ON employee
FOR EACH ROW
BEGIN
   DBMS_OUTPUT.PUT_LINE('Salary updated from ' || :OLD.salary || ' to ' ||
:NEW.salary);
END;
```

- **:OLD** → previous value
- **:NEW** → new value
- Useful for **auditing changes**

INSTEAD OF Trigger (for a view)

```
CREATE OR REPLACE TRIGGER trg_instead_of_update
INSTEAD OF UPDATE ON emp_view
FOR EACH ROW
BEGIN
   UPDATE employee
   SET salary = :NEW.salary
   WHERE employee_id = :OLD.employee_id;
END;
```

- Allows **updating a complex view**
- DML on view is replaced by trigger logic

Real-Life Usage

- HR → Validate salaries or update audit table automatically
- Sales → Track updates in orders or inventory
- Education → Keep log of grade changes
- Banking → Prevent invalid transactions

Interview Questions

Q1. Difference between BEFORE and AFTER trigger?

- BEFORE → executes **before DML**, can prevent action
- AFTER → executes **after DML**, cannot prevent action

Q2. What is INSTEAD OF trigger?

- Executes **in place of DML** on a view, used to modify views

Q3. Can a trigger fire for multiple events?

- Yes, e.g., AFTER INSERT OR UPDATE OR DELETE

Q4. Difference between :NEW and :OLD?

- :NEW → new row values being inserted/updated
- :OLD → existing row values before update/delete

Common Fresher Mistakes

- Forgetting **FOR EACH ROW** for row-level triggers
- Confusing **:NEW and :OLD**
- Trying to modify **:OLD values** → not allowed
- Infinite loops → trigger updating same table without care

Best Practices

- Keep triggers **simple and fast**
- Use **row-level triggers** only when needed
- Avoid DML on same table inside AFTER triggers → infinite loop risk
- Use triggers for **validation, logging, auditing**
- Comment triggers for clarity

PL/SQL Predefined Exceptions (Oracle 26ai)

Definition:

- **Predefined exceptions** → Standard PL/SQL errors automatically **defined by Oracle.**
- Triggered when **specific runtime errors** occur, like dividing by zero or no data found.

Think: “Oracle already knows these common errors, so it gives you ready-to-use exceptions to handle them.”

Why It’s Important

- Prevents program **crashes at runtime**
- Allows **controlled error handling**
- Essential for **fresher interviews** and **robust PL/SQL coding**

Key Points

Exception	Triggered By	Example Usage
NO_DATA_FOUND	SELECT returns **no rows**	Fetching an employee that doesn’t exist

Exception	Triggered By	Example Usage
TOO_MANY_ROWS	SELECT returns **more than 1 row**	Fetching by non-unique criteria
ZERO_DIVIDE	Division by zero	v := 10/0;
VALUE_ERROR	Invalid datatype conversion	Converting string to number fails
OTHERS	Any **unhandled exception**	Catch-all for unexpected errors

- Predefined exceptions **don't need declaration**
- Use **EXCEPTION block** to handle them
- Use **RAISE** to propagate manually if needed

Syntax Example

```
DECLARE
   v_salary NUMBER;
BEGIN
   SELECT salary INTO v_salary
   FROM employee
   WHERE employee_id = 999; -- ID not present

   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee not found!');
   WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE('Multiple employees found!');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Some other error occurred.');
END;
```

- **NO_DATA_FOUND** handles missing row
- **TOO_MANY_ROWS** handles multiple rows
- **OTHERS** handles unexpected errors

Real-Life Usage

- HR → Fetch employee info safely
- Sales → Retrieve order details without crashing
- Banking → Avoid errors in calculations (divide by zero)
- Automation → Handle unexpected runtime errors gracefully

Interview Questions

Q1. Difference between NO_DATA_FOUND and TOO_MANY_ROWS?

- NO_DATA_FOUND → SELECT returned 0 rows
- TOO_MANY_ROWS → SELECT returned >1 row

Q2. Do you need to declare predefined exceptions?

- No, Oracle already defines them

Q3. What is OTHERS exception?

- Catch-all for **any unhandled exceptions**

Q4. Can you raise a predefined exception manually?

- Yes, use RAISE ZERO_DIVIDE; etc.

Common Fresher Mistakes

- Forgetting **EXCEPTION block** → runtime crash
- Handling OTHERS **without logging actual error** → debugging hard
- Misusing ZERO_DIVIDE → doesn't catch numeric overflow
- Using NO_DATA_FOUND on **UPDATE or DELETE** → only for SELECT INTO

Best Practices

- Always handle **common exceptions explicitly**
- Use **OTHERS** to catch unexpected errors
- Log error message using SQLCODE and SQLERRM
- Keep **EXCEPTION blocks clean** and meaningful
- Avoid suppressing exceptions silently

PL/SQL User-Defined Exceptions (Oracle 26ai)

Definition:

- **User-Defined Exception** → A custom exception **created by the developer** to handle **business-specific or custom error conditions** that Oracle predefined exceptions do not cover.

Think: "When Oracle's built-in exceptions aren't enough, you define your own rules for errors."

Why It's Important

- Enforces **business rules in the database**
- Provides **clear, meaningful error messages**
- Crucial for **fresher interviews and real-world PL/SQL programming**

Key Points

Feature	Description
Declaration	Must declare exception in **DECLARE section**
Raising	Use **RAISE** keyword to trigger it
Handling	Catch in **EXCEPTION block**
Optional message	Can use RAISE_APPLICATION_ERROR to give a **custom error code and message**
Scope	Procedure, function, or PL/SQL block

Syntax

```
DECLARE
   my_exception EXCEPTION;  -- declare user-defined exception
BEGIN
   -- some logic
   IF some_condition THEN
      RAISE my_exception;   -- raise exception
   END IF;

EXCEPTION
   WHEN my_exception THEN
      DBMS_OUTPUT.PUT_LINE('Custom exception occurred!');
END;
```

Example 1: Simple User-Defined Exception

```
DECLARE
   insufficient_salary EXCEPTION;
   v_salary NUMBER := 5000;
BEGIN
   IF v_salary < 10000 THEN
```

```
      RAISE insufficient_salary;  -- raise custom exception
   END IF;

   DBMS_OUTPUT.PUT_LINE('Salary is sufficient.');

EXCEPTION
   WHEN insufficient_salary THEN
      DBMS_OUTPUT.PUT_LINE('Salary is below minimum required!');
END;
```

Output:

```
Salary is below minimum required!
```

Example 2: Using RAISE_APPLICATION_ERROR

```
DECLARE
   low_salary EXCEPTION;
BEGIN
   IF 5000 < 10000 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Salary cannot be less than 10000!');
   END IF;
END;
```

- **-20001 to -20999** → Allowed custom error codes
- Displays **custom message** instead of default Oracle message

---

Real-Life Usage

- HR → Raise exception if employee salary < minimum wage
- Banking → Prevent withdrawal if balance < minimum
- Education → Raise exception if marks entered > max limit
- Inventory → Prevent stock from going negative

Interview Questions

Q1. Difference between Predefined and User-Defined Exceptions?

- Predefined → Oracle provides
- User-Defined → Developer creates

Q2. What is RAISE_APPLICATION_ERROR?

- Allows sending **custom error code and message** to calling environment

Q3. Allowed error code range for RAISE_APPLICATION_ERROR?

- -20001 to -20999

Q4. Can user-defined exceptions be caught in a procedure?

- Yes, define **EXCEPTION block** inside procedure

Common Fresher Mistakes

- Forgetting **EXCEPTION block** → unhandled exception error
- Using invalid **error codes in RAISE_APPLICATION_ERROR**
- Not initializing or checking condition before raising exception
- Raising predefined exceptions mistakenly instead of custom

Best Practices

- Use **meaningful names** for exceptions
- Keep **custom error messages clear and precise**
- Always handle exceptions properly to avoid program crash
- Use **RAISE_APPLICATION_ERROR** for communicating errors to applications
- Avoid raising exceptions unnecessarily in loops → performance impact

---

PL/SQL Cursors (Oracle 26ai)

Definition:

- **Cursor** → A pointer that allows **row-by-row processing of query results** in PL/SQL.
- Used when **SELECT returns multiple rows** and you want to handle them **one at a time.**

Think: "A cursor is like a bookmark that points to a specific row in a query result so you can process each row carefully."

Why It's Important

- Enables **processing multiple rows** in PL/SQL blocks
- Avoids **bulk operations errors**
- Commonly asked in **fresher interviews**

Key Points

Feature	Description
Cursor Types	Implicit, Explicit
Implicit Cursor	Automatically created for **DML statements** (INSERT, UPDATE, DELETE, SELECT INTO)
Explicit Cursor	Declared by user for **multi-row SELECT queries**
Attributes	%FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN

Cursor Attributes

Attribute	Meaning
%FOUND	TRUE if last fetch returned a row
%NOTFOUND	TRUE if last fetch did NOT return a row
%ROWCOUNT	Number of rows fetched so far
%ISOPEN	TRUE if cursor is open

Implicit Cursor Example

```
DECLARE
   v_salary employee.salary%TYPE;
BEGIN
   SELECT salary INTO v_salary
   FROM employee
   WHERE employee_id = 101;

   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee not found');
END;
```

- Automatically managed by Oracle
- Best for **single-row SELECT INTO**

Explicit Cursor Example

🔹 **Declaration**

```
DECLARE
   CURSOR emp_cursor IS
      SELECT employee_id, salary FROM employee WHERE department_id = 10;
   v_employee_id employee.employee_id%TYPE;
```

```
   v_salary employee.salary%TYPE;
BEGIN
   OPEN emp_cursor;  -- open cursor

   LOOP
      FETCH emp_cursor INTO v_employee_id, v_salary;  -- fetch row
      EXIT WHEN emp_cursor%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || v_employee_id || ', Salary: ' || v_salary);
   END LOOP;

   CLOSE emp_cursor;  -- close cursor
END;
```

- **OPEN** → initialize cursor
- **FETCH** → retrieve row into variables
- **EXIT WHEN %NOTFOUND** → stop when no rows remain
- **CLOSE** → release resources

Cursor FOR Loop (Simpler Way)

```
BEGIN
   FOR rec IN (SELECT employee_id, salary FROM employee WHERE department_id = 10) LOOP
      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || rec.employee_id || ', Salary: ' ||
rec.salary);
   END LOOP;
END;
```

- No need to **OPEN, FETCH, CLOSE** explicitly
- Ideal for **row-by-row processing**

Real-Life Usage

- HR → Process all employees in a department
- Sales → Compute total sales per employee/order
- Education → Calculate grades for all students
- Inventory → Update stock levels row by row

Interview Questions

Q1. Difference between Implicit and Explicit Cursors?

- Implicit → automatic, single-row DML/SELECT INTO
- Explicit → user-declared, multiple rows

Q2. What are cursor attributes?

- %FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN

Q3. Can you update data using a cursor?

- Yes, with **cursor FOR UPDATE**

Q4. Why use cursor FOR loop?

- Simplifies **row-by-row processing** without manual OPEN/FETCH/CLOSE

Common Fresher Mistakes

- Forgetting to **CLOSE explicit cursor** → memory leak
- Using **FETCH outside loop** incorrectly
- Not checking **%NOTFOUND** before processing
- Confusing **implicit and explicit cursor use**

Best Practices

- Prefer **cursor FOR loop** unless you need more control
- Always **close explicit cursors**
- Use **%ROWCOUNT** to check processed rows

- Avoid **nested cursors** if possible → performance hit
- Use meaningful **cursor and variable names**

PL/SQL Cursors: Implicit vs Explicit (Oracle 26ai)

Definition:

- **Cursor** → A pointer to the **result set of a query** that allows row-by-row processing in PL/SQL.
- **Implicit Cursor** → Automatically created by Oracle for **single-row queries (SELECT INTO) or DML statements.**
- **Explicit Cursor** → Declared by the **developer** to process **multiple-row queries.**

Think: "Implicit cursors are automatic bookmarks, explicit cursors are ones you manually control."

Why It's Important

- Allows **controlled processing of query results**
- Explicit cursors needed when **SELECT returns multiple rows**
- Implicit cursors simplify **single-row processing**
- Frequently asked in **fresher PL/SQL interviews**

Key Differences

Feature	Implicit Cursor	Explicit Cursor
Declaration	Not required, automatically created	Must declare using CURSOR keyword
Use Case	Single-row queries (SELECT INTO) or DML	Multi-row queries requiring row-by-row processing
Control	Limited control	Full control: OPEN, FETCH, CLOSE
Cursor Attributes	%FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN	Same attributes available
Example	SELECT salary INTO v_sal FROM employee WHERE emp_id=101;	CURSOR c_emp IS SELECT * FROM employee;

Implicit Cursor Example

```
DECLARE
   v_salary employee.salary%TYPE;
BEGIN
   SELECT salary INTO v_salary
   FROM employee
   WHERE employee_id = 101;

   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);

   -- Cursor attributes
   IF SQL%FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Row found!');
   END IF;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee not found!');
```

```
END;
```
- Oracle automatically opens, fetches, and closes the cursor
- %FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN can be used

Explicit Cursor Example

```
DECLARE
   CURSOR emp_cursor IS
      SELECT employee_id, salary FROM employee WHERE department_id = 10;

   v_emp_id employee.employee_id%TYPE;
   v_salary employee.salary%TYPE;
BEGIN
   OPEN emp_cursor;  -- open cursor

   LOOP
      FETCH emp_cursor INTO v_emp_id, v_salary;  -- fetch one row
      EXIT WHEN emp_cursor%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || v_emp_id || ', Salary: ' || v_salary);
   END LOOP;

   CLOSE emp_cursor;  -- close cursor
END;
```
- Manual control of cursor needed: **OPEN → FETCH → CLOSE**
- Ideal for **multi-row processing**

Cursor Attributes (for both types)

Attribute	Meaning
%FOUND	TRUE if last fetch returned a row
%NOTFOUND	TRUE if last fetch did NOT return a row
%ROWCOUNT	Number of rows processed so far
%ISOPEN	TRUE if cursor is currently open (explicit only)

Real-Life Usage
- HR → Process salaries of all employees in a department
- Sales → Calculate total sales per salesperson
- Education → Assign grades to multiple students
- Inventory → Update stock row by row

Interview Questions

Q1. Difference between Implicit and Explicit Cursor?
- Implicit → automatic, limited control, single-row queries
- Explicit → developer-defined, full control, multi-row queries

Q2. When do you use implicit cursor?
- Single-row SELECT INTO or DML statements

Q3. When do you use explicit cursor?
- Multiple rows need **row-by-row processing**

Q4. Cursor attributes available?
- %FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN

Common Fresher Mistakes
- Forgetting to **CLOSE explicit cursor** → memory leaks
- Using %ISOPEN for implicit cursor → invalid
- Fetching without checking %NOTFOUND → loop runs indefinitely

- Confusing **implicit and explicit cursors**

Best Practices

- Prefer **implicit cursors** for single-row queries
- Use **cursor FOR loop** for simpler explicit cursor processing
- Always **close explicit cursors**
- Use **meaningful variable names**
- Avoid nested explicit cursors unless necessary

PL/SQL Cursor FOR Loops (Oracle 26ai)

Definition:

- **Cursor FOR Loop** → A **simplified way to process each row** of a query using an **explicit or implicit cursor.**
- Automatically **opens, fetches, and closes** the cursor.

Think: "It's like a magic loop that goes through each row of your query without manually managing the cursor."

Why It's Important

- Simplifies **row-by-row processing**
- Avoids **manual OPEN, FETCH, CLOSE**
- Reduces **errors in cursor handling**
- Common topic in **PL/SQL interviews for freshers**

Key Points

- Cursor variable declared **inside the FOR loop** automatically
- Each row fetched **as a record variable**
- Can be used with **explicit queries** or **named cursors**
- No need to explicitly **open or close cursor**

Syntax

Implicit Cursor FOR Loop

```
FOR record_variable IN (SELECT column1, column2 FROM table_name) LOOP
   -- process each row
   DBMS_OUTPUT.PUT_LINE('Col1: ' || record_variable.column1 || ', Col2: ' ||
record_variable.column2);
END LOOP;
```

Explicit Cursor FOR Loop

```
DECLARE
   CURSOR emp_cursor IS
      SELECT employee_id, salary FROM employee WHERE department_id = 10;
BEGIN
   FOR emp_rec IN emp_cursor LOOP
      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || emp_rec.employee_id || ', Salary: ' ||
emp_rec.salary);
   END LOOP;
END;
```

- **emp_rec** → automatically declared record variable holding one row
- No need for **OPEN, FETCH, CLOSE**
- Efficient and cleaner than traditional explicit cursor loop

---

Real-Life Usage

- HR → Print employee details in a department
- Sales → Calculate total sales for all orders
- Education → List students' grades in a class
- Inventory → Update stock quantities row by row

Interview Questions

Q1. Difference between Cursor FOR Loop and normal explicit cursor?

- FOR Loop → automatically manages OPEN, FETCH, CLOSE
- Explicit cursor → manual OPEN, FETCH, CLOSE

Q2. Can you modify table data inside a Cursor FOR Loop?

- Yes, but careful of **mutating table errors**

Q3. What is the type of record variable in FOR Loop?

- Implicitly a **%ROWTYPE** for the cursor query

Q4. Can Cursor FOR Loop be nested?

- Yes, nested loops possible for multi-level data processing

Common Fresher Mistakes

- Trying to **open/close cursor manually** in FOR loop → redundant
- Modifying same table being queried → mutating table error
- Using wrong column names in record variable
- Confusing record variable with table variable

Best Practices

- Prefer **Cursor FOR Loops** for **row-by-row processing**
- Use **meaningful record variable names**
- Avoid complex operations inside the loop → performance issues
- Keep logic simple and readable
- Combine with **conditional statements** for filtering within loop

---

PL/SQL Parameterized Cursors & REF Cursors (Oracle 26ai)

Parameterized Cursors

Definition:

- A **Parameterized Cursor** is an explicit cursor that **accepts parameters** to make the query **dynamic based on input values.**
- Allows **flexible, reusable cursor logic.**

Think: "It's a cursor that can filter rows based on the input you give it."

Key Points

- Parameters declared in **cursor declaration**
- Can be used in **WHERE clause or calculations**
- Helps avoid multiple similar cursors for different conditions

---

Syntax

```
DECLARE
   CURSOR emp_cursor (p_dept_id NUMBER) IS
      SELECT employee_id, salary
      FROM employee
      WHERE department_id = p_dept_id;

   v_emp_id employee.employee_id%TYPE;
   v_salary employee.salary%TYPE;
```

```
BEGIN
   OPEN emp_cursor(10);  -- pass parameter
   LOOP
      FETCH emp_cursor INTO v_emp_id, v_salary;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || v_emp_id || ', Salary: ' || v_salary);
   END LOOP;
   CLOSE emp_cursor;
END;
```

- **p_dept_id** → cursor parameter
- Allows **dynamic query based on input**

Real-Life Usage

- HR → Get employees for a particular department
- Sales → List orders for a specific customer
- Education → Fetch students for a particular class

REF Cursors (Cursor Variables)
Definition

- **REF Cursor** → A **pointer to a query result set** that can be **passed between PL/SQL blocks or programs.**
- Can be **opened dynamically at runtime.**
- Often used to **return query results to applications (like Java/PLSQL apps).**

Think: “It’s like a universal cursor you can pass around or return from procedures/functions.”

Key Points

- Declared as **REF CURSOR type**
- Can be **strongly typed** (specific row structure) or **weakly typed** (any structure)
- Useful in **modular programming & application interfaces**

Syntax: Weakly Typed REF Cursor

```
DECLARE
   TYPE ref_cursor_type IS REF CURSOR;
   emp_ref ref_cursor_type;
   v_emp_id employee.employee_id%TYPE;
   v_salary employee.salary%TYPE;
BEGIN
   OPEN emp_ref FOR SELECT employee_id, salary FROM employee WHERE department_id = 10;

   LOOP
      FETCH emp_ref INTO v_emp_id, v_salary;
      EXIT WHEN emp_ref%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || v_emp_id || ', Salary: ' || v_salary);
   END LOOP;

   CLOSE emp_ref;
END;
```

Strongly Typed REF Cursor

```
DECLARE
   TYPE emp_ref_type IS REF CURSOR RETURN employee%ROWTYPE;
   emp_ref emp_ref_type;
   v_emp employee%ROWTYPE;
BEGIN
   OPEN emp_ref FOR SELECT * FROM employee WHERE department_id = 10;
```

```
   LOOP
      FETCH emp_ref INTO v_emp;
      EXIT WHEN emp_ref%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || v_emp.employee_id || ', Salary: ' ||
v_emp.salary);
   END LOOP;

   CLOSE emp_ref;
END;
```

Real-Life Usage

- Return query results from **stored procedures/functions** to applications
- Pass cursor results between **PL/SQL blocks**
- Dynamic reports in HR, Sales, Finance, Inventory

Interview Questions

Q1. Difference between Parameterized Cursor and REF Cursor?

- Parameterized Cursor → Accepts input parameters, fixed structure
- REF Cursor → Can be **dynamic**, passed between programs

Q2. What is a weakly typed REF cursor?

- Can return **any query structure**, flexible but less strict

Q3. What is a strongly typed REF cursor?

- Must return **specific table/row structure**, safer for compile-time checks

Q4. Can REF Cursor be returned from a function?

- Yes, commonly used to return **query result sets** to applications

---

Common Fresher Mistakes

- Not closing REF cursors → memory leaks
- Using strong REF cursor with **mismatched query structure**
- Passing parameters incorrectly in parameterized cursors
- Forgetting %NOTFOUND check in loops

Best Practices

- Always **close cursors** (explicit & REF)
- Use **parameterized cursors** for reusable queries
- Use **strong REF cursors** for compile-time safety
- Use **weak REF cursors** for dynamic reports or variable query structures
- Keep logic **modular and readable**

PL/SQL EXECUTE IMMEDIATE (Oracle 26ai)

Definition:

- **EXECUTE IMMEDIATE** → A PL/SQL statement used to **execute dynamic SQL or PL/SQL statements at runtime.**
- Allows queries, DML, DDL, or anonymous PL/SQL blocks to be **constructed as strings** and executed on the fly.

Think: “It’s like telling Oracle, ‘Here’s a SQL string – execute it now!’”

Why It’s Important

- DDL commands (CREATE, ALTER, DROP) **cannot be executed directly in PL/SQL blocks** → need dynamic SQL

- Supports **dynamic queries** based on variables
- Used in **modular code and applications**
- Frequently asked in **interviews**

Key Points

Feature	Description
Dynamic SQL	SQL constructed **at runtime**
Single Row or Multi Row	Can execute queries returning single values or DML
Bind Variables	Use USING clause to prevent SQL injection
DDL Execution	Allows CREATE, DROP, ALTER inside PL/SQL

Syntax

```
EXECUTE IMMEDIATE dynamic_sql_string;
```

- Optional INTO → for single-row SELECT
- Optional USING → for bind variables

Examples

1. DDL Example (Create Table)

```
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE temp_emp(emp_id NUMBER, emp_name VARCHAR2(50))';
END;
```

- Creates table at **runtime**

---

2. DML Example (Insert with Bind Variable)

```
DECLARE
   v_id NUMBER := 101;
   v_name VARCHAR2(50) := 'Ali';
BEGIN
   EXECUTE IMMEDIATE 'INSERT INTO employee(employee_id, employee_name) VALUES(:1, :2)'
   USING v_id, v_name;

   COMMIT;
END;
```

- :1, :2 → placeholders for variables
- USING clause binds PL/SQL variables safely

---

3. SELECT INTO Example

```
DECLARE
   v_salary NUMBER;
   v_emp_id NUMBER := 101;
BEGIN
   EXECUTE IMMEDIATE 'SELECT salary FROM employee WHERE employee_id = :id'
   INTO v_salary
   USING v_emp_id;

   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
END;
```

- Retrieves **single-row value dynamically**

---

4. Drop Table Example

```
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE temp_emp';
END;
```

- Dynamic DDL execution

Real-Life Usage

- Create **audit or temp tables dynamically**
- Insert/update/delete based on **dynamic table names**
- Execute **dynamic queries** in reports and applications
- Handle **schema changes** or modular database updates

Interview Questions

Q1. Difference between static SQL and dynamic SQL?

- Static → SQL fixed at compile time
- Dynamic → SQL constructed and executed at runtime using EXECUTE IMMEDIATE

Q2. Can you execute DDL statements in PL/SQL?

- Only via **dynamic SQL (EXECUTE IMMEDIATE)**

Q3. What is the purpose of USING clause?

- Bind PL/SQL variables to dynamic SQL → prevents **SQL injection**

Q4. Can EXECUTE IMMEDIATE fetch multiple rows?

- No, for multi-row, use **OPEN-FOR dynamic cursor**

Common Fresher Mistakes

- Using dynamic SQL without **bind variables** → SQL injection risk
- Forgetting **INTO** for single-row SELECT
- Using EXECUTE IMMEDIATE for multi-row SELECT → runtime error
- Not committing after DML inside dynamic SQL

Best Practices

- Always use **bind variables** with DML
- Use dynamic SQL **only when necessary**
- Close cursors for **OPEN-FOR dynamic queries**
- Keep SQL string readable → avoid concatenation errors
- Test dynamic SQL carefully before deployment

PL/SQL Error Handling & Logging (Oracle 26ai)

Definition:

- **Error Handling** → The process of **detecting and managing exceptions** (runtime errors) in PL/SQL to prevent program crashes.
- **Logging** → Recording **error details** (like error code, message, timestamp) into a **table or file** for auditing/debugging.

Think: "Errors are inevitable; PL/SQL lets you **catch them and write them down** for review later."

Why It's Important

- Ensures **robust and reliable programs**
- Helps in **troubleshooting and auditing**
- Mandatory for **enterprise applications and Cognizant interview scenarios**

Key Points

Aspect	Description
EXCEPTION Block	Used to **catch exceptions** in PL/SQL
Predefined Exceptions	Oracle-defined (NO_DATA_FOUND, TOO_MANY_ROWS, ZERO_DIVIDE, etc.)

Aspect	Description
User-Defined Exceptions	Developer-created exceptions
SQLCODE & SQLERRM	Functions to get **error code and error message**
Logging Table	Store **error info with timestamp** for later analysis

---

Syntax: Basic Error Handling

```
BEGIN
   -- some logic
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No data found!');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLCODE || ' - ' || SQLERRM);
END;
```

- **NO_DATA_FOUND** → handles missing data
- **OTHERS** → catch-all for unexpected errors
- **SQLCODE** → numeric error code
- **SQLERRM** → error message

Logging Errors into a Table

Create Logging Table

```
CREATE TABLE error_log (
   log_id NUMBER GENERATED ALWAYS AS IDENTITY,
   err_code NUMBER,
   err_msg VARCHAR2(4000),
   err_time TIMESTAMP
);
```

PL/SQL Block with Logging

```
BEGIN
   -- Example: divide by zero
   DECLARE
      v_num NUMBER := 10;
      v_den NUMBER := 0;
      v_result NUMBER;
   BEGIN
      v_result := v_num / v_den;
   EXCEPTION
      WHEN OTHERS THEN
         INSERT INTO error_log(err_code, err_msg, err_time)
         VALUES (SQLCODE, SQLERRM, SYSTIMESTAMP);
         COMMIT;
         DBMS_OUTPUT.PUT_LINE('Error logged successfully.');
   END;
END;
```

- Captures **error code, message, timestamp**
- Useful for **auditing, troubleshooting, or production debugging**

Real-Life Usage

- HR → Log invalid employee updates
- Banking → Track failed transactions
- Sales → Record order entry errors
- Automation → Maintain audit trail for failed scripts

Interview Questions

Q1. Difference between SQLCODE and SQLERRM?

- SQLCODE → Returns numeric error code
- SQLERRM → Returns error message string

Q2. What is OTHERS exception used for?

- Catch-all for **unhandled exceptions**

Q3. Why use error logging?

- To **analyze, debug, and audit** runtime issues without crashing programs

Q4. Can you log user-defined exceptions?

- Yes, use **RAISE or RAISE_APPLICATION_ERROR** and log in error table

---

Common Fresher Mistakes

- Forgetting **EXCEPTION block** → runtime crash
- Logging **without COMMIT** → entries lost
- Using **OTHERS without SQLCODE/SQLERRM** → insufficient info
- Ignoring **error logging in procedures/functions**

Best Practices

- Always handle **common exceptions explicitly**
- Use **logging table with timestamp and details**
- Include **SQLCODE & SQLERRM** in logs
- Commit after logging to persist error records
- Avoid suppressing errors silently → hampers debugging

PL/SQL Bulk Processing (BULK COLLECT & FORALL) (Oracle 26ai)

Definition:

- **Bulk Processing** → Technique in PL/SQL to **process multiple rows at once**, improving performance.
- **BULK COLLECT** → Fetch **multiple rows into collections** in a single operation.
- **FORALL** → Execute **DML statements for multiple rows** in collections efficiently.

Think: “Instead of processing rows one by one, bulk processing handles many rows in one go.”

Why It’s Important

- **Reduces context switches** between PL/SQL and SQL engines
- **Improves performance** for large datasets
- Essential for **enterprise apps and Cognizant fresher interviews**

Key Points

Feature	Description
BULK COLLECT	Fetch multiple rows into a **collection (array, nested table, VARRAY)**
FORALL	Perform **INSERT/UPDATE/DELETE** on multiple rows in a collection at once
Collections	Must declare **nested table, VARRAY, or associative array**
LIMIT clause	Fetch rows in **batches** to avoid memory overflow

---

BULK COLLECT Syntax

```
DECLARE
   TYPE emp_tab IS TABLE OF employee.employee_id%TYPE;
   emp_ids emp_tab;
```

```
BEGIN
   SELECT employee_id BULK COLLECT INTO emp_ids
   FROM employee
   WHERE department_id = 10;

   FOR i IN 1..emp_ids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || emp_ids(i));
   END LOOP;
END;
```

- Fetches all employee IDs **at once** into collection emp_ids

---

FORALL Syntax (Bulk DML)

```
DECLARE
   TYPE emp_tab IS TABLE OF employee.employee_id%TYPE;
   emp_ids emp_tab := emp_tab(101, 102, 103);  -- collection
   new_salary NUMBER := 5000;
BEGIN
   FORALL i IN 1..emp_ids.COUNT
      UPDATE employee
      SET salary = new_salary
      WHERE employee_id = emp_ids(i);

   COMMIT;
END;
```

- Executes **UPDATE** for all employee IDs in one **context switch**
- Much faster than row-by-row updates

BULK COLLECT with LIMIT Clause (Batch Processing)

```
DECLARE
   TYPE emp_tab IS TABLE OF employee%ROWTYPE;
   emp_data emp_tab;
BEGIN
   LOOP
      SELECT * BULK COLLECT INTO emp_data
      FROM employee
      WHERE ROWNUM <= 100;  -- fetch 100 rows at a time

      EXIT WHEN emp_data.COUNT = 0;

      FORALL i IN 1..emp_data.COUNT
         UPDATE employee
         SET salary = salary + 500
         WHERE employee_id = emp_data(i).employee_id;

      COMMIT;
   END LOOP;
END;
```

- Processes **large tables in chunks**
- Prevents **memory overflow**

Real-Life Usage

- HR → Increase salaries for all employees in a department
- Banking → Update interest rates for multiple accounts
- Sales → Apply discounts to multiple orders
- Inventory → Adjust stock for multiple items in bulk

Interview Questions

Q1. Difference between BULK COLLECT and FORALL?

- BULK COLLECT → fetch multiple rows into a collection
- FORALL → perform DML operations on a collection in bulk

Q2. What is the advantage of bulk processing?

- Reduces **context switches**, improves performance for large datasets

Q3. What is the use of LIMIT in BULK COLLECT?

- Fetch **rows in batches** to avoid memory issues

Q4. Can FORALL be used with BULK COLLECT?

- Yes, typically use BULK COLLECT to fetch and FORALL to update

---

Common Fresher Mistakes

- Forgetting **COMMIT** after FORALL → no DML changes persisted
- Not handling **memory overflow** for large tables
- Using **row-by-row loops** instead of bulk → performance hit
- Miscounting **collection index** → runtime error

Best Practices

- Always **use BULK COLLECT with LIMIT** for large tables
- Use **FORALL for bulk DML** instead of looping row by row
- Avoid unnecessary fetching of columns → select only needed columns
- Keep **collections small for memory efficiency**
- Combine with **exception handling** to log errors in bulk operations

---

PL/SQL BULK COLLECT & FORALL (Oracle 26ai)

Definition:

- **BULK COLLECT** → Fetches **multiple rows from SQL queries into PL/SQL collections** in one operation.
- **FORALL** → Executes **DML statements (INSERT, UPDATE, DELETE)** for all elements of a collection in one go.

Think: “Instead of processing rows one by one, BULK COLLECT fetches them in bulk, and FORALL updates/inserts them in bulk.”

Why It’s Important

- Reduces **context switches** between SQL and PL/SQL engines
- Improves **performance for large datasets**
- Essential for **enterprise apps & Cognizant fresher interviews**

Key Points

Feature	BULK COLLECT	FORALL
Purpose	Fetch multiple rows into a collection	Perform DML on all elements of a collection
Data Type	Works with **nested tables, VARRAYs, associative arrays**	Works with **collections only**
Performance	Fetch all rows at once → fewer context switches	Execute DML in bulk → faster than row-by-row
Limit Clause	Optional → fetch rows in batches for memory efficiency	N/A

BULK COLLECT Syntax

```
DECLARE
   TYPE emp_tab IS TABLE OF employee.employee_id%TYPE;
   emp_ids emp_tab;
BEGIN
   SELECT employee_id BULK COLLECT INTO emp_ids
   FROM employee
   WHERE department_id = 10;

   FOR i IN 1..emp_ids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Emp ID: ' || emp_ids(i));
   END LOOP;
END;
```

- Fetches **all employee IDs** for a department in one operation

---

FORALL Syntax

```
DECLARE
   TYPE emp_tab IS TABLE OF employee.employee_id%TYPE;
   emp_ids emp_tab := emp_tab(101, 102, 103);
   new_salary NUMBER := 5000;
BEGIN
   FORALL i IN 1..emp_ids.COUNT
      UPDATE employee
      SET salary = new_salary
      WHERE employee_id = emp_ids(i);

   COMMIT;
END;
```

- Performs **UPDATE** for all employee IDs in the collection **at once**
- Much faster than individual row updates

BULK COLLECT with LIMIT Clause

```
DECLARE
   TYPE emp_tab IS TABLE OF employee%ROWTYPE;
   emp_data emp_tab;
BEGIN
   LOOP
      SELECT * BULK COLLECT INTO emp_data
      FROM employee
      WHERE ROWNUM <= 100;  -- fetch 100 rows at a time

      EXIT WHEN emp_data.COUNT = 0;

      FORALL i IN 1..emp_data.COUNT
         UPDATE employee
         SET salary = salary + 500
         WHERE employee_id = emp_data(i).employee_id;

      COMMIT;
   END LOOP;
END;
```

- Processes **large tables in batches**
- Prevents **memory overflow**

Real-Life Usage

- HR → Update salaries for all employees in a department
- Sales → Apply discounts for multiple orders at once
- Banking → Update interest rates for many accounts
- Inventory → Adjust stock quantities in bulk

Interview Questions

Q1. Difference between BULK COLLECT and FORALL?

- BULK COLLECT → Fetch multiple rows into a collection
- FORALL → Perform DML on collection elements

Q2. Advantage of using BULK COLLECT and FORALL?

- Reduces **context switches**, improves **performance**

Q3. Why use LIMIT clause with BULK COLLECT?

- To fetch **rows in manageable batches**, avoid memory issues

Q4. Can FORALL be combined with BULK COLLECT?

- Yes, fetch rows with BULK COLLECT → update them using FORALL

Common Fresher Mistakes

- Forgetting **COMMIT** after FORALL
- Not using **LIMIT** for large data → memory overflow
- Using **row-by-row loops** instead of bulk operations
- Miscounting collection indexes in FORALL loops

Best Practices

- Use **BULK COLLECT + FORALL** for performance-critical operations
- Always **commit after bulk DML**
- Use **LIMIT clause** for large datasets
- Select only **required columns** to save memory
- Combine with **exception handling** for error logging

JDBC Architecture (Java Database Connectivity)

Definition:

- **JDBC** → A Java API to **connect and interact with relational databases** like Oracle, MySQL, etc.
- Allows Java applications to **execute SQL queries, updates, and retrieve results.**

Think: “JDBC is a bridge between Java programs and databases.”

Why It’s Important

- Core part of **Java backend development**
- Needed for **Spring Boot, Servlets, JSP** integration with DB
- Common **interview topic for freshers**

Key Points

- JDBC provides **standard API** for relational database access
- Uses **drivers** to communicate with specific databases
- Supports **CRUD operations, transaction management, and error handling**

JDBC Architecture Components

Layer / Component	Description
Java Application	Your Java code (JSP/Servlet/Spring Boot) using JDBC API

Layer / Component	Description
JDBC API	Interface classes for **connecting, executing queries, processing results**
JDBC Driver Manager	Manages **database drivers** and selects the appropriate driver
JDBC Driver	Vendor-specific implementation that **translates JDBC calls into DB commands**
Database	Actual **RDBMS (Oracle/MySQL/PostgreSQL)** that stores data

---

Architecture Diagram

```
Java Application
       |
       v
   JDBC API
       |
       v
Driver Manager
       |
       v
JDBC Driver (Oracle/MySQL)
       |
       v
     Database
```

- **Application → JDBC API → Driver Manager → Driver → Database**
- Transparent to programmer, simplifies DB interaction

JDBC Drivers Types

Type	Description	Use Case
Type 1: JDBC-ODBC Bridge	Connects via ODBC	Legacy, not recommended
Type 2: Native API Driver	Uses database vendor native libraries	Medium performance
Type 3: Network Protocol Driver	Translates JDBC calls to database-independent network protocol	Good for multi-tier apps
Type 4: Thin Driver (Pure Java)	Directly converts JDBC calls to DB protocol	Most widely used (Oracle/MySQL)

Steps to Use JDBC

1. **Load Driver** → Class.forName("oracle.jdbc.driver.OracleDriver");
2. **Establish Connection** → DriverManager.getConnection(url, user, password);
3. **Create Statement** → Statement stmt = conn.createStatement();
4. **Execute Query / Update** → ResultSet rs = stmt.executeQuery(query);
5. **Process Results** → while(rs.next()){...}
6. **Close Resources** → rs.close(); stmt.close(); conn.close();

Real-Life Usage

- Banking → Fetch account info, update balances
- E-commerce → Retrieve product details, update orders
- HR → Employee management systems
- Inventory → Stock management, reporting

Interview Questions

Q1. What is JDBC?

- Java API to interact with **relational databases**

Q2. What are JDBC drivers?

- Classes that **translate JDBC calls into database-specific commands**

Q3. Difference between Statement, PreparedStatement, CallableStatement?

- Statement → simple SQL execution
- PreparedStatement → precompiled, prevents SQL injection
- CallableStatement → execute stored procedures

Q4. What is the role of DriverManager?

- Loads and **manages JDBC drivers** to connect with databases

---

Common Fresher Mistakes

- Not closing **ResultSet, Statement, Connection** → memory leaks
- Using **Statement instead of PreparedStatement** → SQL injection risk
- Wrong **driver type** or DB URL
- Ignoring **SQLException handling**

Best Practices

- Always use **PreparedStatement** for dynamic queries
- Close **all JDBC resources** in finally block or use **try-with-resources**
- Use **connection pooling** for performance
- Catch and log **SQLExceptions** properly
- Keep SQL **database-independent** when possible

JDBC Drivers & DriverManager (Oracle 26ai)

JDBC Drivers

Definition:

- A **JDBC Driver** is a **software component** that enables Java applications to **connect and interact with a specific database.**
- Translates **JDBC API calls** into **database-specific commands.**

Think: “Driver = translator between Java and the database.”

Why It’s Important

- Java is **database-independent**, but each DB has its own protocol
- Driver allows **Java to communicate seamlessly** with Oracle, MySQL, SQL Server, etc.
- Crucial for **backend development and interviews**

Types of JDBC Drivers

Type	Name	Description	Pros / Cons
Type 1	JDBC-ODBC Bridge	Uses ODBC driver to connect	Legacy, slow, platform-dependent
Type 2	Native API driver	Uses vendor-specific library	Medium performance, platform-dependent
Type 3	Network Protocol driver	Converts JDBC calls to DB-independent network protocol	Good for multi-tier apps
Type 4	Thin driver (Pure Java)	Directly converts JDBC calls to DB protocol	Fast, platform-independent, widely used

Best Choice for Oracle: Type 4 “Thin Driver”

Example of Loading a Driver (Oracle)

```
// Load Oracle JDBC driver
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- Loads the **driver class dynamically**
- Required before establishing a connection (in older JDBC versions)

DriverManager

Definition

- **DriverManager** → Manages **all registered JDBC drivers**
- Establishes **connection between Java app and database** using getConnection()

Think: "DriverManager = the manager who picks the right driver to connect to the DB."

Key Points

- Maintains a **list of all available drivers**
- Finds an **appropriate driver for a given DB URL**
- Supports **multiple connections to different databases**

Syntax to Connect Database

```
import java.sql.*;

public class JdbcTest {
    public static void main(String[] args) {
        Connection conn = null;
        try {
            // Load driver
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // Connect to DB
            conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:xe",
                "username", "password"
            );

            System.out.println("Connection successful!");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try { if(conn != null) conn.close(); } catch(Exception e) {}
        }
    }
}
```

- **URL format (Oracle Thin Driver)** → jdbc:oracle:thin:@host:port:SID
- DriverManager **selects the driver** based on URL

Real-Life Usage

- Java programs connecting to **Oracle, MySQL, PostgreSQL**
- Spring Boot apps connecting to **Oracle via JDBC**
- Standalone Java DB tools or reporting tools

Interview Questions

Q1. What is JDBC Driver?

- Software component that **enables Java to interact with DB**

Q2. Types of JDBC Drivers?

- Type 1 → JDBC-ODBC Bridge

- Type 2 → Native API
- Type 3 → Network Protocol
- Type 4 → Thin Driver (Pure Java)

Q3. What is DriverManager?

- Manages **all registered drivers** and establishes DB connections

Q4. Difference between Driver and DriverManager?

- Driver → Translates Java calls to DB commands
- DriverManager → Chooses the **right driver** and manages connection

---

Common Fresher Mistakes

- Forgetting to **load driver class** (older JDBC versions)
- Using **Type 1 driver in production** → slow and platform-dependent
- Wrong DB URL format → connection fails
- Not closing **Connection** → memory leaks

Best Practices

- Use **Type 4 driver (Thin driver)** for Oracle or MySQL
- Always **close Connection, Statement, ResultSet**
- Use **try-with-resources** to avoid resource leaks
- Keep **DB credentials secure**
- Use **connection pooling** in enterprise apps

JDBC Connection Interface (Oracle 26ai)

Definition:

- **Connection Interface** → Represents a **session between a Java application and a database.**
- Used to **execute SQL statements, manage transactions, and interact with the database.**

Think: “Connection = the bridge that keeps Java and DB talking.”

Why It’s Important

- Every JDBC operation requires a **Connection object**
- Essential for **CRUD operations, transactions, and resource management**
- Core for **Spring Boot, Servlets, JSP database interactions**

Key Points

- **Obtained via DriverManager.getConnection()**
- Can execute **SQL queries using Statement, PreparedStatement, or CallableStatement**
- Supports **transaction management** (commit, rollback)
- Must **close connection** to release resources

Creating a Connection

```
import java.sql.*;

public class ConnectionExample {
    public static void main(String[] args) {
        Connection conn = null;
        try {
            // Load Oracle driver
            Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
            // Create connection
            conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:xe",
                "username", "password"
            );

            System.out.println("Connection established!");

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try { if(conn != null) conn.close(); } catch(Exception e) {}
        }
    }
}
```

- conn → instance of **Connection interface**
- Used to **execute SQL statements** and manage **transactions**

Main Methods of Connection Interface

Method	Description
createStatement()	Creates a **Statement object** to execute SQL
prepareStatement(String sql)	Creates a **PreparedStatement** for parameterized queries
prepareCall(String sql)	Creates **CallableStatement** to execute stored procedures
setAutoCommit(boolean flag)	Enable/disable **auto-commit**
commit()	Commit current **transaction**
rollback()	Rollback current **transaction**
close()	Close connection and release resources
isClosed()	Checks if connection is **already closed**
setTransactionIsolation(int level)	Set **isolation level** for transactions

---

Transaction Management Example

```
try {
    conn.setAutoCommit(false);  // disable auto-commit

    Statement stmt = conn.createStatement();
    stmt.executeUpdate("UPDATE employee SET salary = salary + 500 WHERE department_id
= 10");

    conn.commit();  // commit transaction
    System.out.println("Transaction committed successfully");

} catch(SQLException e) {
    conn.rollback();  // rollback on error
    e.printStackTrace();
} finally {
    conn.close();  // close connection
}
```

- setAutoCommit(false) → manual control of transaction
- commit() → saves changes
- rollback() → undoes changes on error

Real-Life Usage

- Bank → Transfer money between accounts (requires commit/rollback)
- HR → Update multiple employee records safely
- E-commerce → Place order transactionally

Interview Questions

Q1. What is Connection interface in JDBC?

- Represents **session between Java app and database**

Q2. How do you obtain a Connection?

- DriverManager.getConnection(url, username, password)

Q3. Difference between auto-commit true/false?

- True → Each SQL executed immediately
- False → Must **commit manually**

Q4. What is the use of rollback()?

- Undo transaction in case of **error or exception**

Common Fresher Mistakes

- Not closing **connection** → memory leaks
- Using auto-commit without thinking → partial data changes
- Forgetting **rollback** on exceptions
- Using **Statement instead of PreparedStatement** → SQL injection risk

---

Best Practices

- Always **close Connection** in finally block or try-with-resources
- Use **PreparedStatement** for dynamic queries
- Manage transactions properly → commit/rollback
- Use **connection pooling** in enterprise apps
- Handle **SQLException** gracefully

JDBC Statement & PreparedStatement (Oracle 26ai)

Definition:

Interface	Definition
Statement	Used to execute **simple SQL queries** without parameters.
PreparedStatement	Used to execute **precompiled SQL queries with parameters**. More secure and faster.

Think:

- Statement → for **fixed queries**
- PreparedStatement → for **dynamic queries with variables**

Why It's Important

- Core part of **JDBC CRUD operations**
- PreparedStatement prevents **SQL injection**
- Both are widely asked in **fresher interviews**

Key Differences

Feature	Statement	PreparedStatement
SQL Compilation	Compiled **every time**	**Precompiled** once
Parameters	Cannot use parameters	Can use **placeholders ?**

Feature	Statement	PreparedStatement
Security	Vulnerable to **SQL injection**	**Safe** from SQL injection
Performance	Slower for repeated queries	Faster for repeated queries
Usage	Simple queries	Dynamic queries, repeated execution

Statement Example

```
import java.sql.*;

public class StatementExample {
    public static void main(String[] args) throws Exception {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:xe", "username", "password"
        );

        Statement stmt = conn.createStatement();
        String sql = "INSERT INTO employee(employee_id, employee_name) VALUES(101,
'Ali')";
        int rows = stmt.executeUpdate(sql);

        System.out.println(rows + " row inserted.");
        stmt.close();
        conn.close();
    }
}
```

- Executes **fixed SQL query**
- Each execution **recompiles SQL**

PreparedStatement Example

```
import java.sql.*;

public class PreparedStatementExample {
    public static void main(String[] args) throws Exception {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:xe", "username", "password"
        );

        String sql = "INSERT INTO employee(employee_id, employee_name) VALUES(?, ?)";
        PreparedStatement pstmt = conn.prepareStatement(sql);

        pstmt.setInt(1, 102);          // set first placeholder
        pstmt.setString(2, "Sara");    // set second placeholder
        int rows = pstmt.executeUpdate();

        System.out.println(rows + " row inserted.");
        pstmt.close();
        conn.close();
    }
}
```

- Uses **placeholders ?**
- **Safe from SQL injection**
- Efficient for **repeated executions with different parameters**

Real-Life Usage

- Statement → Execute **static queries**, e.g., select all products
- PreparedStatement → Execute **dynamic queries**, e.g., insert orders, update salaries based on user input
- Enterprise apps → Almost all dynamic queries use PreparedStatement

---

Interview Questions

Q1. Difference between Statement and PreparedStatement?

- Statement → Fixed SQL, compiled every time, less secure
- PreparedStatement → Precompiled, parameterized, faster, secure

Q2. Why use PreparedStatement?

- **Prevent SQL injection**
- **Improves performance** for repeated queries

Q3. Can Statement execute parameterized queries?

- No, Statement cannot use placeholders (?)

Q4. Which is faster for repeated execution?

- **PreparedStatement**, because it is precompiled

Common Fresher Mistakes

- Using Statement for user input → **SQL injection risk**
- Forgetting to **close Statement/PreparedStatement**
- Using executeQuery for DML → should use executeUpdate
- Setting wrong **parameter index** in PreparedStatement

Best Practices

- Prefer **PreparedStatement over Statement**
- Always **close resources** (ResultSet, Statement, Connection)
- Use **batch processing** for multiple DML with PreparedStatement
- Handle **SQLException** gracefully
- Use **try-with-resources** for automatic cleanup

JDBC CallableStatement (Oracle 26ai)

Definition:

- **CallableStatement** → A JDBC interface used to **call stored procedures or functions** in the database from Java.
- Can handle **IN, OUT, and INOUT parameters**.

Think: “CallableStatement = the bridge to execute pre-defined database logic from Java.”

Why It’s Important

- Used in **enterprise applications** to encapsulate logic in the database
- Helps in **reusing stored procedures**
- Supports **complex operations like payroll, banking transactions, or batch jobs**
- Frequently asked in **fresher interviews**

Key Points

Feature	Description
IN parameter	Pass value from Java to procedure
OUT parameter	Retrieve value from procedure to Java
INOUT parameter	Pass value in and get modified value out

Feature	Description
Precompiled execution	Stored procedure is precompiled in DB → fast execution
SQL Injection safe	Parameters are bound, not concatenated

Syntax

```
CallableStatement cstmt = conn.prepareCall("{call procedure_name(?, ?)}");
```

- {call procedure_name(?, ?)} → placeholders for parameters
- Can register **OUT parameters** using registerOutParameter()
- Set **IN parameters** using setXXX() methods

Example 1: Procedure with IN Parameter

Stored Procedure in Oracle

```
CREATE OR REPLACE PROCEDURE update_salary(
   emp_id IN NUMBER,
   increment IN NUMBER
)
AS
BEGIN
   UPDATE employee SET salary = salary + increment WHERE employee_id = emp_id;
   COMMIT;
END;
```

Java Code

```
CallableStatement cstmt = conn.prepareCall("{call update_salary(?, ?)}");
cstmt.setInt(1, 101);  // emp_id
cstmt.setInt(2, 500);  // increment
cstmt.executeUpdate();
System.out.println("Salary updated successfully");
cstmt.close();
```

Example 2: Procedure with OUT Parameter

Stored Procedure

```
CREATE OR REPLACE PROCEDURE get_salary(
   emp_id IN NUMBER,
   emp_salary OUT NUMBER
)
AS
BEGIN
   SELECT salary INTO emp_salary FROM employee WHERE employee_id = emp_id;
END;
```

Java Code

```
CallableStatement cstmt = conn.prepareCall("{call get_salary(?, ?)}");
cstmt.setInt(1, 101);                 // IN parameter
cstmt.registerOutParameter(2, Types.NUMERIC);  // OUT parameter
cstmt.execute();
int salary = cstmt.getInt(2);         // Retrieve OUT value
System.out.println("Salary: " + salary);
cstmt.close();
```

Example 3: Procedure with INOUT Parameter

Stored Procedure

```
CREATE OR REPLACE PROCEDURE adjust_bonus(
   emp_id IN NUMBER,
   bonus IN OUT NUMBER
)
```

```
AS
BEGIN
   bonus := bonus + 500;  -- increment bonus
END;
```

Java Code

```
CallableStatement cstmt = conn.prepareCall("{call adjust_bonus(?, ?)}");
cstmt.setInt(1, 101);                 // IN parameter
cstmt.setInt(2, 1000);                // Initial bonus
cstmt.registerOutParameter(2, Types.NUMERIC);
cstmt.execute();
int newBonus = cstmt.getInt(2);       // Retrieve updated bonus
System.out.println("New Bonus: " + newBonus);
cstmt.close();
```

Real-Life Usage

- Banking → Calculate interest, update account balance
- HR → Update salaries, bonuses, or leave balances
- Sales → Generate monthly commission reports
- Inventory → Adjust stock in bulk

Interview Questions

Q1. What is CallableStatement?

- Interface to **execute stored procedures/functions** in JDBC

Q2. Difference between Statement, PreparedStatement, and CallableStatement?

- Statement → Simple fixed SQL
- PreparedStatement → Parameterized SQL
- CallableStatement → Calls **stored procedures/functions**

Q3. What is IN, OUT, INOUT parameter?

- IN → Input to procedure
- OUT → Output from procedure
- INOUT → Input and output

Q4. How do you retrieve an OUT parameter?

- Use registerOutParameter() and then getXXX() method

Common Fresher Mistakes

- Not registering OUT parameters → runtime error
- Mixing **IN and OUT parameter indices**
- Forgetting to close CallableStatement
- Using executeUpdate() for procedures returning values → use execute()

Best Practices

- Always **close CallableStatement**
- Use **parameterized calls** to prevent SQL injection
- Use **transactions with commit/rollback** if procedure updates data
- Test procedure in **SQL Developer** before calling from Java
- Handle **SQLException** properly

JDBC ResultSet Interface (Oracle 26ai)

Definition:

- **ResultSet** → A JDBC interface that **stores the result of a SQL query** executed by a Statement or PreparedStatement.

- Allows **reading, navigating, and retrieving data** from database tables in Java.

Think: “ResultSet = the container holding your query results for Java to read.”

Why It’s Important

- Central to **querying data from databases in Java**
- Supports **row-wise and column-wise access**
- Essential for **CRUD operations, reports, and Cognizant fresher interviews**

Key Points

Feature	Description
Navigation	Move through rows using next(), previous(), first(), last()
Data Retrieval	Use getInt(), getString(), getDate() etc. to fetch column values
Type	Can be **Forward-only** or **Scrollable**
Concurrency	Can be **Read-only** or **Updatable**
Cursor	ResultSet maintains a **pointer (cursor)** to current row

---

Creating a ResultSet

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT employee_id, employee_name, salary FROM
employee");

while(rs.next()) {
    int id = rs.getInt("employee_id");
    String name = rs.getString("employee_name");
    double salary = rs.getDouble("salary");
    System.out.println(id + " | " + name + " | " + salary);
}

rs.close();
stmt.close();
```

- rs.next() → Moves cursor to **next row**, returns false at end
- Columns can be accessed by **name** or **index**

ResultSet Types

Type	Description
TYPE_FORWARD_ONLY	Cursor moves **only forward** (default)
TYPE_SCROLL_INSENSITIVE	Can move forward/backward, **does not reflect DB changes**
TYPE_SCROLL_SENSITIVE	Cursor **reflects DB changes** dynamically

Syntax for Scrollable ResultSet:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);
ResultSet rs = stmt.executeQuery("SELECT * FROM employee");

rs.last();
System.out.println("Last Employee ID: " + rs.getInt("employee_id"));
```

ResultSet Concurrency

Type	Description
CONCUR_READ_ONLY	Cannot update database through ResultSet
CONCUR_UPDATABLE	Can update DB directly via ResultSet

Updatable ResultSet Example:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE
);
ResultSet rs = stmt.executeQuery("SELECT * FROM employee");

rs.next();
rs.updateDouble("salary", 6000);
rs.updateRow();  // Updates database
```

Navigation Methods

- next() → Move forward
- previous() → Move backward
- first() → Go to first row
- last() → Go to last row
- absolute(int row) → Go to specific row
- relative(int rows) → Move relative to current row

Real-Life Usage

- HR → Display employee details
- Banking → Show transaction history
- E-commerce → List orders or products
- Reporting → Generate dynamic reports in Java apps

Interview Questions

Q1. What is ResultSet in JDBC?

- Holds **query results**, allows Java to **read and update database rows**

Q2. Difference between TYPE_FORWARD_ONLY and TYPE_SCROLL_INSENSITIVE?

- Forward-only → cursor moves forward only
- Scroll-insensitive → cursor can move both ways, DB changes not reflected

Q3. How to retrieve column values?

- rs.getInt("column_name"), rs.getString(index), rs.getDate() etc.

Q4. Can ResultSet update the database?

- Yes, if **CONCUR_UPDATABLE** is used with scrollable ResultSet

---

Common Fresher Mistakes

- Forgetting to **call rs.next()** before accessing data
- Using **column index incorrectly**
- Not closing **ResultSet & Statement** → memory leaks
- Trying to **update read-only ResultSet**

Best Practices

- Always **close ResultSet and Statement**
- Use **PreparedStatement** for dynamic queries
- Use **column names instead of indexes** for readability
- Use **scrollable & updatable ResultSet** only when necessary
- Handle **SQLException** properly

Executing Queries & Batch Processing in JDBC (Oracle 26ai)

Executing Queries in JDBC
Definition:

- JDBC provides methods to **execute SQL statements**:
  - **DML (INSERT, UPDATE, DELETE)**
  - **DDL (CREATE, ALTER, DROP)**
  - **SELECT queries**

Think: "Execution methods tell JDBC how to run your SQL and what kind of result to expect."

Key Methods in Statement / PreparedStatement / CallableStatement

Method	Use Case	Returns
executeQuery(sql)	For **SELECT statements**	ResultSet
executeUpdate(sql)	For **INSERT, UPDATE, DELETE, DDL**	int (affected rows)
execute(sql)	Any SQL (DML, DDL, or SELECT)	boolean (true if ResultSet returned)

Example: SELECT

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT employee_id, employee_name FROM employee");

while(rs.next()) {
    System.out.println(rs.getInt("employee_id") + " | " +
rs.getString("employee_name"));
}
rs.close();
stmt.close();
```

- executeQuery → retrieves **data**
- ResultSet → used to navigate & read rows

Example: INSERT / UPDATE

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("UPDATE employee SET salary = salary + 500 WHERE
department_id = 10");
System.out.println(rows + " rows updated.");
stmt.close();
```

- executeUpdate → returns number of **affected rows**

Batch Processing in JDBC
Definition

- **Batch processing** → Execute **multiple SQL statements in a single network round-trip.**
- Improves performance by **reducing database communication overhead.**

Think: "Instead of sending queries one by one, send them in a batch → faster execution."

Why It's Important

- Handles **large data operations efficiently**
- Widely used in **bulk inserts, updates, and deletions**
- Common in **enterprise apps & interviews**

Steps for Batch Processing

1. Create **Statement or PreparedStatement**
2. Add SQL statements using addBatch()
3. Execute all statements with executeBatch()
4. Commit transaction (optional)

Example: Statement Batch

```
Statement stmt = conn.createStatement();
conn.setAutoCommit(false);  // optional

stmt.addBatch("INSERT INTO employee(employee_id, employee_name) VALUES(201, 'Ali')");
stmt.addBatch("INSERT INTO employee(employee_id, employee_name) VALUES(202, 'Sara')");
stmt.addBatch("UPDATE employee SET salary = salary + 500 WHERE employee_id = 101");

int[] results = stmt.executeBatch();  // execute all at once
conn.commit();  // commit transaction

System.out.println("Batch executed, rows affected: " + results.length);
stmt.close();
```

Example: PreparedStatement Batch

```
String sql = "INSERT INTO employee(employee_id, employee_name) VALUES(?, ?)";
PreparedStatement pstmt = conn.prepareStatement(sql);

pstmt.setInt(1, 301);
pstmt.setString(2, "John");
pstmt.addBatch();

pstmt.setInt(1, 302);
pstmt.setString(2, "Emma");
pstmt.addBatch();

int[] results = pstmt.executeBatch();
System.out.println("Batch executed: " + results.length + " rows");
pstmt.close();
```

- **Safe from SQL injection**
- Ideal for **dynamic bulk operations**

Transaction Management with Batch

```
try {
    conn.setAutoCommit(false); // manual commit

    Statement stmt = conn.createStatement();
    stmt.addBatch("INSERT INTO employee(employee_id, employee_name) VALUES(401,
'Ali')");
    stmt.addBatch("UPDATE employee SET salary = salary + 500 WHERE employee_id =
101");

    stmt.executeBatch();
    conn.commit(); // commit all changes
} catch(SQLException e) {
    conn.rollback(); // rollback if any error
    e.printStackTrace();
}
```

- Ensures **all-or-nothing execution**

- Prevents **partial updates**

Real-Life Usage

- Banking → Bulk update of account balances
- HR → Insert multiple employee records at once
- E-commerce → Update inventory for multiple products
- Reporting → Insert large datasets into analytics tables

Interview Questions

Q1. Difference between executeQuery, executeUpdate, execute?

- executeQuery → SELECT, returns ResultSet
- executeUpdate → INSERT/UPDATE/DELETE, returns affected rows
- execute → Any SQL, returns boolean

Q2. What is batch processing in JDBC?

- Execute **multiple SQL statements together** for performance

Q3. Advantages of batch processing?

- Reduces **network round-trips**
- **Faster execution** for bulk operations

Q4. Can PreparedStatement be used in batch?

- Yes, ideal for **dynamic and parameterized batch operations**

Common Fresher Mistakes

- Forgetting addBatch() → only last statement executes
- Using executeUpdate() for multiple statements → slow
- Not committing transactions in batch → changes lost
- Not handling **BatchUpdateException** properly

Best Practices

- Use **PreparedStatement for dynamic batches**
- Always **commit after batch** or use try-catch with rollback
- Limit batch size for **large datasets** to avoid memory issues
- Handle **SQLException and BatchUpdateException** gracefully

---

JDBC Transaction Management (Oracle 26ai)

Definition:

- **Transaction** → A sequence of one or more **SQL operations** executed as a **single unit of work.**
- **Transaction management** → Controlling **commit, rollback, and savepoints** in JDBC to ensure **data consistency.**

Think: “Transaction = all-or-nothing execution of multiple database operations.”

Why It’s Important

- Ensures **ACID properties** (Atomicity, Consistency, Isolation, Durability)
- Prevents **partial updates** in case of errors
- Critical in **banking, payroll, and inventory applications**
- Frequently asked in **fresher interviews**

Key Points

Concept	Description
Auto-commit mode	If true (default), each SQL executes **immediately**
Manual commit	Set conn.setAutoCommit(false) to control transactions manually
Commit	Save all changes permanently → conn.commit()
Rollback	Undo changes if an error occurs → conn.rollback()
Savepoint	Intermediate points within a transaction to rollback partially → conn.setSavepoint()

Steps for Transaction Management

1. Disable auto-commit → conn.setAutoCommit(false)
2. Execute SQL statements (INSERT, UPDATE, DELETE)
3. Commit if all succeed → conn.commit()
4. Rollback if error → conn.rollback()

Example: Basic Transaction

```
Connection conn = null;
try {
    conn = DriverManager.getConnection(dbUrl, username, password);
    conn.setAutoCommit(false);  // manual commit

    Statement stmt = conn.createStatement();
    stmt.executeUpdate("UPDATE account SET balance = balance - 500 WHERE account_id =
101");
    stmt.executeUpdate("UPDATE account SET balance = balance + 500 WHERE account_id =
102");

    conn.commit();  // commit all changes
    System.out.println("Transaction committed successfully");

} catch(SQLException e) {
    if(conn != null) conn.rollback();  // rollback if error
    e.printStackTrace();
} finally {
    if(conn != null) conn.close();
}
```

- Transfers money **atomically** between accounts
- Prevents **partial transfer**

Example: Using Savepoints

```
Connection conn = DriverManager.getConnection(dbUrl, username, password);
conn.setAutoCommit(false);

Savepoint sp1 = null;
try {
    Statement stmt = conn.createStatement();

    stmt.executeUpdate("INSERT INTO employee(employee_id, employee_name) VALUES(201,
'Ali')");
    sp1 = conn.setSavepoint("AfterFirstInsert");

    stmt.executeUpdate("INSERT INTO employee(employee_id, employee_name) VALUES(202,
'Sara')");
```

```
      // Suppose error occurs
    // Rollback to first insert
    conn.rollback(sp1);

    conn.commit();
    System.out.println("Transaction committed with savepoint rollback");

} catch(SQLException e) {
    conn.rollback(); // rollback entire transaction
    e.printStackTrace();
} finally {
    conn.close();
}
```

- Savepoint → allows **partial rollback** without undoing entire transaction

---

Real-Life Usage

- Banking → Transfer funds between accounts
- HR → Update multiple employee records safely
- E-commerce → Process orders and payment in one transaction
- Inventory → Adjust stock counts and logs together

Interview Questions

Q1. What is transaction management in JDBC?

- Control **commit, rollback, and savepoints** to maintain data consistency

Q2. Difference between auto-commit true and false?

- True → each SQL executes immediately
- False → manual commit, all SQLs execute as a single transaction

Q3. What is a Savepoint?

- Intermediate point to **rollback partially** within a transaction

Q4. What are ACID properties?

- Atomicity, Consistency, Isolation, Durability → ensure reliable transactions

---

Common Fresher Mistakes

- Not disabling **auto-commit** → no control over transactions
- Forgetting to **rollback on exceptions**
- Misusing **Savepoints**
- Not committing after successful transactions
- Closing connection before commit → changes lost

Best Practices

- Disable **auto-commit** for multi-step operations
- Always **commit or rollback** explicitly
- Use **savepoints** for complex transactions
- Handle **SQLExceptions** properly
- Close **Connection, Statement, ResultSet** in finally block or try-with-resources

---

JDBC Connection Pooling (Oracle 26ai)

Definition:

- **Connection Pooling** → Technique of **creating a pool of reusable database connections** and sharing them among multiple clients instead of creating a new connection every time.

Think: “Connection pool = a bank of ready-to-use connections to avoid opening a new connection every time.”

Why It’s Important

- **Creating a DB connection is expensive** in terms of time and resources
- Pooling **improves performance** and reduces server load
- Essential in **enterprise apps, Spring Boot, and Cognizant training interviews**

Key Points

- Reduces **connection creation and closing overhead**
- Maintains a **fixed number of connections** in the pool
- Provides **ready-to-use connections** to multiple threads
- Supports **scaling for concurrent requests**

How Connection Pooling Works

1. Initialize a **pool of connections** at application startup
2. Client requests a connection → connection given from pool
3. After use, connection returned to pool (not closed)
4. Pool manages **max connections, idle timeout, and reuse**

Example Using Apache DBCP (Basic)

```
import java.sql.*;
import org.apache.commons.dbcp2.BasicDataSource;

public class ConnectionPoolExample {
    public static void main(String[] args) throws Exception {
        // Create connection pool
        BasicDataSource ds = new BasicDataSource();
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        ds.setUsername("username");
        ds.setPassword("password");
        ds.setMinIdle(5);
        ds.setMaxIdle(10);
        ds.setMaxOpenPreparedStatements(50);

        // Get connection from pool
        Connection conn = ds.getConnection();

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM employee");
        while(rs.next()) {
            System.out.println(rs.getInt("employee_id") + " | " +
rs.getString("employee_name"));
        }

        // Return connection to pool
        conn.close();  // NOT actually closed, returned to pool
        rs.close();
        stmt.close();
    }
}
```

- conn.close() → returns connection to pool instead of closing
- Pool manages **efficient reuse of connections**

Advantages

- **Performance** → Avoid creating/closing connections repeatedly
- **Scalability** → Supports high concurrent DB requests
- **Resource management** → Limits max open connections
- **Reduces latency** → Faster response for client requests

Real-Life Usage

- Web applications with **high traffic** (banking, e-commerce)
- Spring Boot → uses **HikariCP** by default
- Enterprise apps → multiple users querying the database concurrently
- Reporting systems → bulk queries executed efficiently

Interview Questions

Q1. What is connection pooling?

- A pool of **pre-created reusable connections** shared among multiple clients

Q2. Why is it used?

- To **improve performance** and reduce overhead of creating/closing connections

Q3. How is it implemented in Java?

- Using libraries like **Apache DBCP, HikariCP, C3P0**

Q4. Difference between closing connection in normal JDBC vs pool?

- Normal JDBC → connection is **closed**
- Connection pool → connection is **returned to pool for reuse**

Common Fresher Mistakes

- Closing connection in pool → pool mismanagement
- Not configuring **min/max pool size** → poor performance
- Not handling **exceptions** when obtaining connection
- Using connection pool for very small, one-time scripts → unnecessary overhead

---

Best Practices

- Use **connection pooling in enterprise apps**
- Always **return connection to pool** using close()
- Configure **minIdle, maxIdle, maxTotal** properly
- Handle **SQLExceptions** during connection retrieval
- Prefer **HikariCP** for Spring Boot apps (default, high performance)
