**PL/SQL Control Structures, Exception Handling, and Advanced Constructs**

[CLICK HERE -](https://mindmup-export.s3.us-east-1.amazonaws.com/map.png/out/43b84be0577f11f09cacc553efb4aed4.map.png?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-PAYLOAD&X-Amz-Credential=ASIASNCK5ADR7MNCFSRZ%2F20250702%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20250702T200055Z&X-Amz-Expires=93600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEPr%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLWVhc3QtMSJHMEUCIB%2BDqCbVQDafvEjWInLAKSB8Aul6%2BjeE6lhZsQez50qqAiEAvfvT3q5PcRYtjPYDEApABOzlPeroKIyU3NVYIMvdD4gq%2FQII8%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FARAAGgwxNjU1MTMzMzA5MTUiDK0DrOmYeLx7AxKTnyrRAr6OGCCSuozlI5dWDGEMyCCg7re6b8KFdyeULpt1V6LHfVbrZ8p76oklo7ua%2BkvCHGeKhvD2cnLMxUaSugkkyDxEeZXeo%2FINt56qvwsZBX9WfEQkCnYM%2FEICMPrM5teKbH%2F%2Fa4Lf5Ed5ALiHHhGJkWAQc2%2BgIXWBizWZ8kTLnxeq5XQKbV%2B1u8OiGCzjzfUdfUvpFYK%2FHFVH5g1SzY0MSlIUJNCVnXvITPLjiXRU6hA6kTZ0vXUOodBB8fPcLFGoLyxhzUryi7VjMNzO3P2d0CVHGfhOGpZkFBXCM2qlBZaTuCQX4QPUSHL0chOnLPSyJtOrcYqJryXmkI8e4QSaC%2BECouC9R%2B9vBqHKi1Tac6C4C0JpxsCneGrPE6zyntb0bo1MvkFrOZOIjLX3OE%2Bek2WrpQYetdu41F2XKMNsqS0u66tuUCjBOyM%2BdxF90yrTE8gwouKVwwY6ngFt%2FKPdC1UPHH6Mkla0RuHLNiwPhQgrLgZ0E4nHenwMwMav4S0zrEEJ8SG5CFhLb7G4Gi3m94WnWxCqU7HQMe17HLidSYz9mBvYPiZCVPhHlMWy2qAEfZ6uzN09wNlTonkIPARUFYjheOWt92ltL9GlNg2xPd0727RXD%2Bd3YaiPcGxKPHrTFDpkGfi2PGz%2B%2BKakUDcBK47kBUysM5zwsg%3D%3D&X-Amz-Signature=19bc5756607b659cc2d4bf62a684d9fdada8c11d6684e5f00e3d5dc6db4af842&X-Amz-SignedHeaders=host&x-amz-checksum-mode=ENABLED&x-id=GetObject)

<https://app.mindmup.com/map/_free/2025/07/2905bb70577f11f095aa11fc6b2b74ae>

<https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/plsql-collections-and-records.html#GUID-8060F01F-B53B-48D4-9239-7EA8461C2170>

### **PL/SQL Topics – Summary Table**

| **Topic** | **Definition** | **Syntax Example** | **Real Use Case Scenarios** |
| --- | --- | --- | --- |
| **Scalar & Anchored Variables** | Single-value storage; anchored types use column datatype. | v\_salary employees.salary%TYPE; | 1. Store current timestamp 2. Capture employee salary 3. Hold department code 4. Use %TYPE for safe insertions 5. Store derived calculation (bonus) |
| **Associative Array** | In-memory key-value collection (INDEX BY). | TYPE map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER; | 1. Store department heads 2. Map subject-code to name 3. Count login attempts 4. Track temp ticket assignments 5. Daily sales mapped to day no |
| **Nested Table** | Variable-size, index-based collection (can be stored in tables). | TYPE t\_marks IS TABLE OF NUMBER; | 1. Collect survey answers 2. Store dynamic list of orders 3. Return result sets from function 4. Log payment batches 5. Save inventory SKU IDs |
| **VARRAY** | Fixed-size, ordered collection. | TYPE name\_list IS VARRAY(5) OF VARCHAR2(50); | 1. Store top 5 scorers 2. Daily metrics 3. Recent logins 4. Fixed-set product highlights 5. Last 3 manager actions |
| **RECORD** | Composite type with multiple fields. | TYPE emp\_rec IS RECORD (name VARCHAR2(50), sal NUMBER); | 1. Employee profile snapshot 2. Ticket + status grouped 3. Product + price + category 4. Result summary 5. Transfer form input |
| **OBJECT Type** | User-defined datatype with structure. | CREATE TYPE emp\_obj AS OBJECT (...); | 1. Invoice object 2. Student object 3. Product object 4. Report record 5. Feedback type |
| **Views** | Saved SQL query as a virtual table. | CREATE VIEW dept\_stats AS SELECT ...; | 1. Student GPA report 2. Monthly revenue 3. Open tickets list 4. High-severity logs 5. Employee role-mapping |
| **Inline Views** | Query inside FROM clause. | SELECT \* FROM (SELECT ... ORDER BY ...) WHERE ROWNUM = 1; | 1. Latest transaction 2. Top performer 3. Expiring contract 4. Most expensive product 5. Peak login hour |
| **IF-ELSE / CASE** | Branching control flow. | IF v\_sal > 5000 THEN ... ELSE ... END IF; | 1. Salary grading 2. Mark classification 3. Access role validation 4. Discount logic 5. Language selector |
| **FOR / WHILE Loops** | Repeat operations for known/unknown counts. | FOR i IN 1..5 LOOP ... END LOOP; | 1. Loop through marks 2. Update rows 3. Allocate stock 4. Retry login 5. Print invoice lines |
| **Exception Handling** | Manage runtime errors; PRAGMA for mapping. | WHEN NO\_DATA\_FOUND THEN ... PRAGMA EXCEPTION\_INIT(e\_custom, -20001); | 1. Missing student 2. Duplicate entry 3. Salary < 0 4. Account locked 5. Insufficient stock |
| **Implicit Cursor** | Auto-managed cursor for single DML. | SELECT col INTO var FROM ...; | 1. Single-row fetch 2. Basic validation 3. Count records 4. Fetch config 5. Check login attempt |
| **Explicit Cursor** | Manually defined and controlled. | CURSOR cur IS SELECT ...; | 1. Bulk display 2. Line item loop 3. Salary update 4. Notification list 5. Migration of legacy data |
| **Parameterized Cursor** | Cursor with WHERE clause input. | CURSOR c(p\_dept VARCHAR2) IS SELECT ...; | 1. Students in one dept 2. Tickets for one user 3. Filtered sales 4. Orders by status 5. Tasks per team |

Case Study 1: IT Helpdesk Ticketing System (Comprehensive Real-World Example)

### **Project Context Recap: IT Helpdesk Ticketing System**

This system is designed to manage technical support requests in an organization. Each ticket logs:

* The issue reported
* Its severity
* The assigned engineer
* Its current status

We aim to **simulate real-time ticket data processing** using PL/SQL constructs that are also applied in **production-grade ITSM platforms** like ServiceNow, Freshdesk, or BMC Remedy.

## Objective:

Automate IT ticket logging, categorization, engineer assignment, and status tracking using PL/SQL constructs.

## Components Used:

* Scalar & Anchored Variables
* Collections: Associative Array, Nested Table, VARRAY
* Object Types & Records
* Views & Inline Views
* Control Structures (IF, CASE, Loops)
* Exception Handling: PRAGMA, Predefined, User-defined
* Cursors: Implicit, Explicit, Parameterized

## Step 1: Create Supporting Tables

CREATE TABLE engineers (  
 engineer\_id NUMBER PRIMARY KEY,  
 engineer\_name VARCHAR2(100),  
 department VARCHAR2(50)  
);  
  
CREATE TABLE tickets (  
 ticket\_id NUMBER PRIMARY KEY,  
 issue\_desc VARCHAR2(200),  
 severity CHAR(1),  
 assigned\_to NUMBER,  
 created\_on DATE,  
 status VARCHAR2(20)  
);  
  
CREATE SEQUENCE ticket\_seq START WITH 1001 INCREMENT BY 1;

## Step 2: Insert Data Using Scalar & Anchored Types

DECLARE  
 v\_id NUMBER := ticket\_seq.NEXTVAL;  
 v\_desc tickets.issue\_desc%TYPE := 'System Crash';  
 v\_status VARCHAR2(20) := 'OPEN';  
BEGIN  
 INSERT INTO tickets VALUES (v\_id, v\_desc, 'H', 101, SYSDATE, v\_status);  
 DBMS\_OUTPUT.PUT\_LINE('Ticket created with ID ' || v\_id);  
END;

## Step 3: Object Type, VARRAY, and CASE

ticket\_obj Defined a class-like datatype

ticket\_list Created a fixed-size array of ticket\_obj

ticket\_obj(...) Instantiated individual objects

ticket\_list(...) Used constructor to store those objects in array form

v\_list Is now a fully ready-to-loop list of in-memory tickets

In **Java**, to create and populate a List<Ticket> with new Ticket(...) objects, you would do the following:

### ✅ **Step-by-Step Syntax**

1. **Create the Ticket class** (with constructor):

java

CopyEdit

public class Ticket {

int id;

String description;

String severity;

public Ticket(int id, String description, String severity) {

this.id = id;

this.description = description;

this.severity = severity;

}

}

1. **Declare and Populate the List**:

java

CopyEdit

import java.util.\*;

public class TicketSystem {

public static void main(String[] args) {

List<Ticket> ticketList = new ArrayList<>();

ticketList.add(new Ticket(1011, "VPN Fail", "M"));

ticketList.add(new Ticket(1012, "Email Down", "H"));

ticketList.add(new Ticket(1013, "Laptop Crash", "H"));

// Print details

for (Ticket t : ticketList) {

System.out.println("Ticket ID: " + t.id + ", Desc: " + t.description + ", Severity: " + t.severity);

}

}

}

## ✅ Real-World Analogy

| **Concept** | **PL/SQL** | **Java** |
| --- | --- | --- |
| Blueprint of Ticket | OBJECT TYPE | class Ticket |
| Create 1 ticket | ticket\_obj(...) | new Ticket(...) |
| Create list of 3 tickets | ticket\_list(...) | List<Ticket> with new Ticket(...) added |
| Loop through tickets | FOR i IN 1..COUNT | for (Ticket t : list) |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question : so here the varray is fixed and we are passing the data through object constructor then why we have used

ticket\_list(ticket-obj ) v\_list ticket\_list := ticket\_list( ticket\_obj(1011, 'VPN Fail', 'M'), ticket\_obj(1012, 'Email Down', 'H') ); can you tell me the flow execution of creating object and then creating a fixed varray and then saving the data using object constructor

DECLARE

v\_list ticket\_list := ticket\_list(

ticket\_obj(1011, 'VPN Fail', 'M'),

ticket\_obj(1012, 'Email Down', 'H')

);

🧠 First, Recap: What’s Happening?

You're combining:

An object (ticket\_obj) = a custom blueprint for ticket records

A VARRAY (ticket\_list) = a fixed-size list of those ticket objects

An object constructor call = manually initializing objects to populate the list

✅ Execution Flow – Step by Step

Step What Happens Behind the Scenes

1. Define ticket\_obj type CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (...) Oracle creates a user-defined datatype (like a Java class).

2. Define ticket\_list type CREATE OR REPLACE TYPE ticket\_list IS VARRAY(5) OF ticket\_obj; Oracle sets up a collection type (array of ticket\_objs, max 5 elements).

3. Declare variable v\_list v\_list ticket\_list := ... You declare a variable of type ticket\_list. It's still empty at this point.

4. Use object constructors ticket\_obj(1011, 'VPN Fail', 'M') Each call creates an in-memory object instance with those values.

5. Pass values to VARRAY constructor ticket\_list(...ticket\_obj...) The ticket\_list constructor accepts 1–5 objects and stores them in memory.

6. Final v\_list result v\_list now holds a ready-to-use list of ticket objects Each element has .id, .desc, and .severity accessible like a class property.

🔁 What Does Oracle Do Internally?

Memory Allocation

Oracle allocates memory to hold a ticket\_list of up to 5 objects.

Constructor Call

Each ticket\_obj(...) triggers the object’s constructor — this fills in the .id, .desc, .severity fields.

Fills the VARRAY

Oracle populates the VARRAY in order, storing the object instances at indices 1 and 2.

Loop Execution (if used)

In the loop that follows, Oracle uses .COUNT to find how many elements and allows indexed access.

🧩 Think of it Like This (Analogy)

plsql

Copy

Edit

-- CLASS + ARRAY in one line:

ticket\_list(

ticket\_obj(1011, 'VPN Fail', 'M'),

ticket\_obj(1012, 'Email Down', 'H')

)

Is like writing in Python:

python

Copy

Edit

[

Ticket(1011, 'VPN Fail', 'M'),

Ticket(1012, 'Email Down', 'H')

]

Or in Java:

java

Copy

Edit

new Ticket[] {

new Ticket(1011, "VPN Fail", "M"),

new Ticket(1012, "Email Down", "H")

};

✅ Summary

Concept What You Did

ticket\_obj Defined a class-like datatype

ticket\_list Created a fixed-size array of ticket\_obj

ticket\_obj(...) Instantiated individual objects

ticket\_list(...) Used constructor to store those objects in array form

v\_list Is now a fully ready-to-loop list of in-memory tickets

Would you like a visual diagram of this memory structure or a version that reads from the table and fills the VARRAY using EXTEND? I can prepare both.

You said:

so means we have two constructors here one is of varray and one is of object constructor

ChatGPT said:

✅ Yes — exactly!

You are spot-on. There are two separate constructors at play in that line:

🔧 1. Object Constructor → ticket\_obj(...)

This constructor creates an individual object instance of type ticket\_obj:

plsql

Copy

Edit

ticket\_obj(1011, 'VPN Fail', 'M')

🔹 Creates: one record/row-like object in memory

🔹 Fields set: id, desc, severity

🔹 Use: Acts like a class instance (in OOP)

➡️ Think of it as:

"Create 1 ticket object with these 3 values."

🔧 2. VARRAY Constructor → ticket\_list(...)

This constructor creates a collection (ordered array) of ticket\_obj:

plsql

Copy

Edit

ticket\_list(

ticket\_obj(1011, 'VPN Fail', 'M'),

ticket\_obj(1012, 'Email Down', 'H')

)

🔹 Creates: a VARRAY that can hold up to 5 ticket\_obj objects (as defined)

🔹 Populates: the array with object instances in order

🔹 Use: You can loop through it, access via index, etc.

➡️ Think of it as:

"Create a list of 2 ticket objects and assign them to v\_list."

🧠 Recap in Layers:

Layer Constructor Used Example Description

🧱 Object Layer ticket\_obj(...) ticket\_obj(1011, 'VPN Fail', 'M') Creates a single record

📦 Collection Layer ticket\_list(...) ticket\_list(ticket\_obj(...), ticket\_obj(...)) Creates an array and fills it with objects

✅ Final Result

After this:

plsql

Copy

Edit

v\_list ticket\_list := ticket\_list(

ticket\_obj(1011, 'VPN Fail', 'M'),

ticket\_obj(1012, 'Email Down', 'H')

);

You have an in-memory array that looks like:

Index ID Description Severity

1 1011 VPN Fail M

2 1012 Email Down H

Would you like:

A dry-run with memory visualization?

Or a version that fills the VARRAY dynamically using a cursor?

Happy to help!

You said:

so how many layers we have in pl/sql and the constructor as compare to java

ChatGPT said:

Great follow-up! Let’s unpack this by comparing PL/SQL and Java in terms of:

✅ Layers of Code Execution and Constructs in PL/SQL vs Java

Conceptual Layer PL/SQL Equivalent Java Equivalent

1️⃣ Scalar Values Variables (e.g. v\_name VARCHAR2(50);) Primitive types (e.g. int, double, String)

2️⃣ Record/Struct RECORD, %ROWTYPE Java class or POJO

3️⃣ Object Type OBJECT (e.g. CREATE TYPE ticket\_obj AS OBJECT (...)) Java class (with constructor, fields, etc.)

4️⃣ Collection - VARRAY (fixed-size)

- Nested Table (dynamic)

- Associative Array (map-like) Java Arrays, ArrayList, HashMap, etc.

5️⃣ Collection of Objects VARRAY of OBJECTS, Nested Table of OBJECTS Array or List<MyObject>

6️⃣ Control Blocks BEGIN...END with IF, CASE, LOOP Method bodies ({ ... })

7️⃣ Procedural Units Procedures, Functions Methods

8️⃣ Exception Handling EXCEPTION block, PRAGMA, user-defined try-catch blocks, custom exceptions

9️⃣ Cursors Implicit, Explicit, Parameterized ResultSet iteration using JDBC

🔟 Views & Inline Views SQL views and subqueries Not native in Java — simulated through DAO/result filters

🎯 Constructor Comparison: PL/SQL vs Java

🔹 PL/SQL Constructors

PL/SQL supports constructor-like behavior only in OBJECT TYPES and VARRAYS.

Constructor Type Example Purpose

Object Type Constructor ticket\_obj(1011, 'VPN Fail', 'H') Create a structured data record

VARRAY Constructor ticket\_list(ticket\_obj(...), ticket\_obj(...)) Create a fixed-size collection of objects

⚠️ Note: These are declarative constructors, not code-based like in Java. You can't override them.

CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (  
 id NUMBER,  
 desc VARCHAR2(100),  
 severity CHAR(1)  
);  
  
CREATE OR REPLACE TYPE ticket\_list IS VARRAY(5) OF ticket\_obj;  
  
DECLARE  
 v\_list ticket\_list := ticket\_list(ticket\_obj(1011, 'VPN Fail', 'M'), ticket\_obj(1012, 'Email Down', 'H'));  
BEGIN  
 FOR i IN 1..v\_list.COUNT LOOP  
 CASE v\_list(i).severity  
 WHEN 'H' THEN DBMS\_OUTPUT.PUT\_LINE('High Severity: ' || v\_list(i).desc);  
 ELSE DBMS\_OUTPUT.PUT\_LINE('Other Severity: ' || v\_list(i).desc);  
 END CASE;  
 END LOOP;  
END;

Explanations :

When support teams triage tickets, they often:

* View batches of issues
* Sort or group them by **severity**
* Prioritize high-impact problems like system crashes or network failures

So in our implementation:

* We simulate a **list of recent tickets**
* Use CASE to handle **severity-based logic**
* Structure data cleanly using **object and collection types**

## Why Use **Object Type (**ticket\_obj**)**?

CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (

id NUMBER,

desc VARCHAR2(100),

severity CHAR(1)

);

This defines a **custom ticket datatype** where each record has:

* id (Ticket ID)
* desc (Short description)
* severity (H, M, L — High, Medium, Low)

**Why object?**

* It mirrors real-world ticket entities
* Keeps related fields together like a class in OOP
* Easy to reuse in collections or return from functions

## Why Use **VARRAY (**ticket\_list**)**?

CREATE OR REPLACE TYPE ticket\_list IS VARRAY(5) OF ticket\_obj;

VARRAYs are:

* **Fixed-size**, ordered collections
* Efficient when we know the max ticket count to process (e.g., **latest 5 tickets**)

**Why VARRAY?**

* We’re modeling a limited result set (like dashboard summary)
* Easy to traverse using a loop
* Memory-efficient compared to nested tables for fixed-size use

## Why Use **CASE** in the Loop?

CASE v\_list(i).severity

WHEN 'H' THEN DBMS\_OUTPUT.PUT\_LINE('High Severity: ' || v\_list(i).desc);

ELSE DBMS\_OUTPUT.PUT\_LINE('Other Severity: ' || v\_list(i).desc);

END CASE;

**Why CASE?**

* Handles different business rules based on severity
* Makes the code **scalable** if more logic is added later (e.g., auto-assign H tickets)
* Easier to read and maintain than nested IFs

**It reflects how IT teams actually:**

* Fetch a chunk of tickets
* Group by severity
* Perform actions based on urgency

Question: why create an OBJECT type like ticket\_obj when we already have the same information in the tickets table?

## 1. **Object Types Are Not Replacing Tables — They Complement In-Memory Use**

Think of the table as your **permanent storage** — the database's long-term memory (like a hard disk).  
The **object type** is like a **RAM-based blueprint** for processing small units of that data in memory.

We **don’t always want to query the database repeatedly** — especially when:

* We're dealing with a small, filtered set (e.g., last 5 open tickets)
* We need fast, **temporary**, structured access in PL/SQL block logic

## 2. **Why Not Just Use the Table Directly in the Loop?**

We absolutely can — and we did in other parts of the project using:

FOR r IN (SELECT \* FROM tickets WHERE ...) LOOP ...

But using a table query every time:

* Hits the database again
* Requires more parsing
* May not reflect a **structured or grouped unit** (like an object does)

When we define:

ticket\_obj(id, desc, severity)

We can:

* Treat it like a **unit of data** in collections
* **Group** values cleanly (no need to manage 3 separate variables)
* **Pass it around** to procedures or functions later

## 3. **Real-World Case: Ticket Snapshot Report**

Imagine your application wants to:

* Grab the **last 5 high-severity issues**
* Send them to a dashboard/report/middleware
* Format them in a JSON-like structure

Here’s what we do:

* Extract the top 5 into a **VARRAY of ticket\_obj**
* Format/process them without touching the main tickets table again

This avoids:

* Excessive DB reads
* Repeated joins/filters
* Spaghetti code for looping separate columns

## Summary: Why We Created the ticket\_obj and Used It in VARRAY

| **Reason** | **Explanation** |
| --- | --- |
| **Structured logic** | Group related ticket fields into one object |
| **In-memory processing** | Avoids repeated queries to tickets table |
| **Temporary snapshots** | Good for reporting, API processing, dashboards |
| **Modular, clean code** | Easier to maintain than separate variables |
| **Scalable logic** | Object can grow (e.g., add escalated BOOLEAN) |

are we not collecting the actual data and saving it in object using constructor or we are passing some dummy which is not taking from the table

### ✅ Short Answer:

In this code block:

plsql

CopyEdit

DECLARE

v\_list ticket\_list := ticket\_list(

ticket\_obj(1011, 'VPN Fail', 'M'),

ticket\_obj(1012, 'Email Down', 'H')

);

➡️ We are **not** collecting data from the table.

❗ We are manually creating **dummy records** (hardcoded values) using the **object constructor**:

plsql

CopyEdit

ticket\_obj(1011, 'VPN Fail', 'M')

So this line is saying:

"Create a new object of type ticket\_obj with id = 1011, description = 'VPN Fail', and severity = 'M'."

### 🧱 Why Are We Using Dummy Data Here?

This step is meant to:

* **Demonstrate** how to initialize a VARRAY of OBJECT types
* Show how you can loop through objects and apply logic (like CASE)
* Simulate a batch of tickets without needing to access a table

It’s a **training scenario**, not a production use yet.

### 🧠 So, Can We Populate It from the tickets Table?

**Yes! Absolutely.**  
In a real-world situation, you'd likely use a **cursor** or **BULK COLLECT** to populate the object array like this:

plsql

CopyEdit

DECLARE

v\_list ticket\_list := ticket\_list();

BEGIN

FOR r IN (

SELECT ticket\_id, issue\_desc, severity

FROM tickets

WHERE ROWNUM <= 5

) LOOP

v\_list.EXTEND;

v\_list(v\_list.LAST) := ticket\_obj(r.ticket\_id, r.issue\_desc, r.severity);

END LOOP;

END;

✅ This version reads **actual ticket data** from the table and fills the object collection dynamically.

### 🔄 Summary: Are We Fetching from Table?

| **Scenario** | **Answer** |
| --- | --- |
| Using ticket\_obj(...) in declaration | ❌ No, using **hardcoded dummy data** |
| Reading from tickets table | ✅ Possible, but not in current block |
| Can we combine object + VARRAY + query? | ✅ Yes, using FOR loop and EXTEND |

### 🧠 Breakdown:

| **Java Element** | **Purpose** |
| --- | --- |
| List<Ticket> | Declares a list that can store Ticket objects |
| new Ticket(...) | Instantiates a Ticket object using the constructor |
| add() | Adds the object to the list |
| for-each loop | Iterates over the list to access each Ticket |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Absolutely! Let's revisit **Step 3: Object Type, VARRAY, and CASE** — and this time, refactor it to **read from your actual tickets table** (instead of hardcoding values), and then walk you through advanced use cases like:

* ✅ Returning a VARRAY of objects from a **function**
* ✅ Passing that array to a **procedure**
* ✅ Inserting it into a **table that stores the array** (using a column with VARRAY type)

## 🔁 Original Block (Recap from Step 3)

plsql

CopyEdit

CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (

id NUMBER,

desc VARCHAR2(100),

severity CHAR(1)

);

CREATE OR REPLACE TYPE ticket\_list IS VARRAY(5) OF ticket\_obj;

DECLARE

v\_list ticket\_list := ticket\_list(

ticket\_obj(1011, 'VPN Fail', 'M'),

ticket\_obj(1012, 'Email Down', 'H')

);

BEGIN

FOR i IN 1..v\_list.COUNT LOOP

CASE v\_list(i).severity

WHEN 'H' THEN DBMS\_OUTPUT.PUT\_LINE('High Severity: ' || v\_list(i).desc);

ELSE DBMS\_OUTPUT.PUT\_LINE('Other Severity: ' || v\_list(i).desc);

END CASE;

END LOOP;

END;

➡️ This uses **hardcoded data**, not the live tickets table.

## ✅ Refactored: Pull Live Data from tickets Table into the Object Array

### Step 1: Make Sure Object & VARRAY Type Exist

plsql

CopyEdit

CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (

id NUMBER,

desc VARCHAR2(100),

severity CHAR(1)

);

CREATE OR REPLACE TYPE ticket\_list IS VARRAY(10) OF ticket\_obj;

### Step 2: Live Data Fetch Using Bulk Collect + Constructor

plsql

CopyEdit

DECLARE

v\_list ticket\_list := ticket\_list(); -- initialize empty VARRAY

BEGIN

FOR r IN (

SELECT ticket\_id, issue\_desc, severity

FROM tickets

WHERE ROWNUM <= 5 -- Limit for VARRAY(10)

ORDER BY created\_on DESC

) LOOP

v\_list.EXTEND;

v\_list(v\_list.COUNT) := ticket\_obj(r.ticket\_id, r.issue\_desc, r.severity);

END LOOP;

FOR i IN 1..v\_list.COUNT LOOP

CASE v\_list(i).severity

WHEN 'H' THEN DBMS\_OUTPUT.PUT\_LINE('High Severity: ' || v\_list(i).desc);

WHEN 'M' THEN DBMS\_OUTPUT.PUT\_LINE('Medium Severity: ' || v\_list(i).desc);

ELSE DBMS\_OUTPUT.PUT\_LINE('Low Severity: ' || v\_list(i).desc);

END CASE;

END LOOP;

END;

✅ Now you're using **live ticket records** stored in your tickets table rather than hardcoding values.

## 🧠 Why This Matters?

| **Scenario** | **What You're Doing** |
| --- | --- |
| Object | Representing each ticket as a reusable structured entity |
| VARRAY | Batching tickets in memory (e.g., for a dashboard or reporting block) |
| CASE Logic | Classify or respond differently based on severity |
| Real Data | Dynamically populating from tickets table |

## 📦 Advanced Use Cases

### 🔁 1. Function That Returns a VARRAY of Ticket Objects

plsql

CopyEdit

CREATE OR REPLACE FUNCTION get\_top\_tickets RETURN ticket\_list IS

v\_tickets ticket\_list := ticket\_list();

BEGIN

FOR r IN (SELECT ticket\_id, issue\_desc, severity FROM tickets WHERE ROWNUM <= 5) LOOP

v\_tickets.EXTEND;

v\_tickets(v\_tickets.COUNT) := ticket\_obj(r.ticket\_id, r.issue\_desc, r.severity);

END LOOP;

RETURN v\_tickets;

END;

### 🔂 2. Procedure That Accepts the Array and Logs It

plsql

CopyEdit

CREATE OR REPLACE PROCEDURE log\_ticket\_array(p\_list IN ticket\_list) IS

BEGIN

FOR i IN 1..p\_list.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE(p\_list(i).id || ' - ' || p\_list(i).desc || ' - ' || p\_list(i).severity);

END LOOP;

END;

Call like this:

plsql

CopyEdit

DECLARE

v ticket\_list := get\_top\_tickets();

BEGIN

log\_ticket\_array(v);

END;

## 🗃️ 3. Table That Stores VARRAY of Ticket Objects

### Step 1: Create Table With VARRAY Column

plsql

CopyEdit

CREATE TABLE ticket\_batches (

batch\_id NUMBER PRIMARY KEY,

tickets ticket\_list

);

### Step 2: Insert the Array Into the Table

plsql

CopyEdit

DECLARE

v\_batch ticket\_list := get\_top\_tickets();

BEGIN

INSERT INTO ticket\_batches (batch\_id, tickets) VALUES (1, v\_batch);

END;

You’ve now **persisted a collection of objects** (a VARRAY of ticket objects) into a table!

## 💡 Summary

| **Action** | **Benefit** |
| --- | --- |
| Refactor to read from table | Real-time accuracy |
| Use object constructor | Enforces structure & type |
| Wrap in function | Reusable ticket loaders |
| Pass to procedure | Business logic module |
| Insert into VARRAY column | Store structured data efficiently |
| **Insert multiple new ticket records** into the actual tickets table using:   * An **object type** to define ticket structure * A **VARRAY or Nested Table** to hold the bulk tickets * A **FORALL** statement for efficient bulk insertion  Step-by-Step ImplementationStep 1: Define the Object and Collection Type -- 1. Object Type representing a Ticket  CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (  issue\_desc VARCHAR2(100),  severity CHAR(1),  assigned\_to NUMBER,  created\_on DATE,  status VARCHAR2(20)  );  -- 2. Nested Table Type (for bulk insert use)  CREATE OR REPLACE TYPE ticket\_ntt IS TABLE OF ticket\_obj;  We're using a **nested table type** here because it's ideal for dynamic/bulk operations and usable with FORALL. Step 2: PL/SQL Block to Bulk Insert into tickets Table DECLARE  -- 1. Collection of ticket\_obj  v\_tickets ticket\_ntt := ticket\_ntt(  ticket\_obj('Internet not working', 'H', 101, SYSDATE, 'OPEN'),  ticket\_obj('Email failure', 'M', 102, SYSDATE, 'OPEN'),  ticket\_obj('VPN Login issue', 'L', 103, SYSDATE, 'OPEN')  );  BEGIN  -- 2. Bulk Insert using FORALL  FORALL i IN v\_tickets.FIRST .. v\_tickets.LAST  INSERT INTO tickets (  ticket\_id,  issue\_desc,  severity,  assigned\_to,  created\_on,  status  ) VALUES (  ticket\_seq.NEXTVAL,  v\_tickets(i).issue\_desc,  v\_tickets(i).severity,  v\_tickets(i).assigned\_to,  v\_tickets(i).created\_on,  v\_tickets(i).status  );  DBMS\_OUTPUT.PUT\_LINE('Inserted ' || v\_tickets.COUNT || ' tickets successfully.');  END; |  |

So go to the step 3 ✅ Refactored: Pull Live Data from tickets Table into the Object Array and step 4 and wrap this in a procedure so learners can reuse it with different teams or departments?

###### ChatGPT said:

Absolutely! ✅ Wrapping **Step 3 (object + VARRAY + live data fetch)** and **Step 4 (processing associative array)** into procedures is the **best practice** — it allows:

* 🔁 Reuse by multiple teams (like IT, HR, DevOps)
* 🧱 Better modularity and separation of logic
* 🔒 Controlled access based on parameters (like department ID or assigned engineer)

## 🔧 GOAL

You will get **2 reusable procedures**:

1. ✅ proc\_display\_ticket\_summary:
   * Uses VARRAY of ticket\_obj
   * Fetches top N live tickets from the tickets table
   * Displays ticket description with CASE on severity
2. ✅ proc\_engineer\_map:
   * Uses an associative array
   * Maps engineer names based on engineer IDs
   * Loops and prints all engineers

## ✅ Procedure 1: proc\_display\_ticket\_summary

### 🎯 Use Case:

Let departments like **Networking, IT, DevOps** reuse this to quickly summarize top tickets by severity.

### ✅ Definition:

plsql

CopyEdit

CREATE OR REPLACE PROCEDURE proc\_display\_ticket\_summary (

p\_limit IN NUMBER DEFAULT 5

) AS

v\_list ticket\_list := ticket\_list();

BEGIN

-- Fetch live tickets into the VARRAY

FOR r IN (

SELECT ticket\_id, issue\_desc, severity

FROM tickets

WHERE ROWNUM <= p\_limit

ORDER BY created\_on DESC

) LOOP

v\_list.EXTEND;

v\_list(v\_list.COUNT) := ticket\_obj(r.ticket\_id, r.issue\_desc, r.severity);

END LOOP;

-- Print ticket summary

FOR i IN 1..v\_list.COUNT LOOP

CASE v\_list(i).severity

WHEN 'H' THEN DBMS\_OUTPUT.PUT\_LINE(' High Severity: ' || v\_list(i).desc);

WHEN 'M' THEN DBMS\_OUTPUT.PUT\_LINE('Medium Severity: ' || v\_list(i).desc);

ELSE DBMS\_OUTPUT.PUT\_LINE(' Low Severity: ' || v\_list(i).desc);

END CASE;

END LOOP;

END;

### ▶️ Execution Example:

plsql

CopyEdit

BEGIN

proc\_display\_ticket\_summary(3); -- Fetch top 3 recent tickets

END;

## ✅ Procedure 2: proc\_engineer\_map

### 🎯 Use Case:

HR/Admins or department leads need a list of engineers in order, using an associative array for simplicity.

### ✅ Definition:

plsql

CopyEdit

CREATE OR REPLACE PROCEDURE proc\_engineer\_map IS

TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;

eng\_names eng\_map;

idx PLS\_INTEGER := 1;

BEGIN

FOR r IN (SELECT engineer\_name FROM engineers ORDER BY engineer\_id) LOOP

eng\_names(idx) := r.engineer\_name;

idx := idx + 1;

END LOOP;

FOR i IN 1..eng\_names.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Engineer #' || i || ': ' || eng\_names(i));

END LOOP;

END;

### ▶️ Execution Example:

plsql

CopyEdit

BEGIN

proc\_engineer\_map;

END;

## 🧠 Learning Recap:

| **Concept** | **Procedure** | **Highlights** |
| --- | --- | --- |
| Object Constructor | ticket\_obj(...) | Used to create VARRAY of objects |
| VARRAY Collection | ticket\_list | Fixed size; holds ticket summary |
| CASE Logic | In proc\_display\_ticket\_summary | Classify ticket severity |
| BULK FETCH with Loop | In both procedures | Fetch multiple rows from DB |
| Associative Array | proc\_engineer\_map | Indexed key-value collection |
| Modularity | Both procedures | Reusable by different teams |

## ✅ Want to go further?

Let me know if you want:

* A version that **inserts this object array into a table**
* A function that **returns ticket\_list**
* A scheduler job that runs proc\_display\_ticket\_summary daily

## Step 4: Associative Array for Engineer Mapping

DECLARE  
 TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;  
 eng\_names eng\_map;  
BEGIN  
 SELECT engineer\_name BULK COLLECT INTO eng\_names FROM engineers;  
 DBMS\_OUTPUT.PUT\_LINE('Engineer: ' || eng\_names(1));  
END;

Explanations :

## 👨‍💼 **Project Manager’s Real-World Perspective:**

### 🔹 Business Context:

In our **IT Helpdesk Ticketing System**, we must **dynamically reference engineer names** based on their IDs for ticket assignments, dashboards, and quick lookups.

### Why We Need This:

* In real applications (like in HCL or Infosys ticket systems), you don’t always want to query the DB again and again to get a name for an ID — **performance will suffer**.
* Instead, we **load engineer names once into memory** and then access them in-memory like a dictionary or key-value pair → this is exactly what **associative arrays** help us do in PL/SQL.

### Where It Fits in the Architecture:

* **Memory-resident lookup**: like in-memory cache → no need for repeated DB round-trips.
* Ideal in **batch processing**, **email alerts**, or **ticket report generation** when you need to refer to multiple engineers many times.
* Data read-only after load — ensures consistency during execution.

## 👨‍💻 **Technical PL/SQL Developer Explanation:**

### 🔹 Code:

plsql

CopyEdit

DECLARE

TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;

eng\_names eng\_map;

BEGIN

SELECT engineer\_name BULK COLLECT INTO eng\_names FROM engineers;

DBMS\_OUTPUT.PUT\_LINE('Engineer: ' || eng\_names(1));

END;

### 🔹 Breakdown:

| **Element** | **Description** |
| --- | --- |
| TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER; | Declares an **associative array** (key-value structure). Key is number, value is engineer name. |
| eng\_names eng\_map; | Instantiates the associative array. |
| SELECT ... BULK COLLECT INTO eng\_names | Loads multiple rows at once into memory (**bulk fetch**) — faster than row-by-row. |
| eng\_names(1) | Accesses the value at index 1 (first engineer name). |

## 🧠 Why Use Associative Arrays (Architectural Advantage):

| **Feature** | **Benefit** |
| --- | --- |
| In-memory | Speeds up access and eliminates repetitive DB queries. |
| Index-based | Easily maps IDs to names, or positions to values. |
| Flexible | Can handle **sparse or dynamic data** (e.g., only engineers from a department). |
| Bulk Operations | Works great with BULK COLLECT, reducing context switches. |

### ✅ Real-World Scenario:

You are generating **end-of-day reports** or sending **email alerts** with ticket summaries. Instead of fetching engineer names multiple times from the engineers table, you use an associative array to store them **once**, then reuse them across hundreds of loop iterations.

Would you like me to extend this with:

* engineer\_id as a key (INDEX BY PLS\_INTEGER → mapping engineer\_id → engineer\_name)?
* JSON-style output?
* Integration with ticket assignment logic?

##### You said:

eng\_names(1) Accesses the value at index 1 (first engineer name). it;s giving first row data only how it will give all

###### ChatGPT said:

Great question — you're absolutely right to question this.

### 🧠 Clarification:

Yes, in this example:

plsql

CopyEdit

DBMS\_OUTPUT.PUT\_LINE('Engineer: ' || eng\_names(1));

This **only prints the first engineer's name** — because we're directly accessing index 1.

## ✅ If You Want to Print **All Engineer Names** in the Associative Array:

You need to **loop over all keys** in the array using:

### 🔁 FIRST .. LAST with a WHILE loop:

plsql

CopyEdit

DECLARE

TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;

eng\_names eng\_map;

idx PLS\_INTEGER;

BEGIN

SELECT engineer\_name

BULK COLLECT INTO eng\_names

FROM engineers;

idx := eng\_names.FIRST; -- get the first key

WHILE idx IS NOT NULL LOOP

DBMS\_OUTPUT.PUT\_LINE('Engineer: ' || eng\_names(idx));

idx := eng\_names.NEXT(idx); -- move to the next key

END LOOP;

END;

### 🔍 Explanation:

| **Line** | **Meaning** |
| --- | --- |
| FIRST | Gets the first index (like 1) |
| NEXT(idx) | Finds the next valid index (like 2, then 3, etc.) |
| WHILE ... IS NOT NULL | Ends when there are no more valid indexes |

### ✅ Output (Assuming 3 engineers):

makefile

CopyEdit

Engineer: John Smith

Engineer: Rina Roy

Engineer: David Jacob

### 🔄 Alternative (using FOR i IN 1..eng\_names.COUNT):

This **does NOT work well with associative arrays** — because their index is not guaranteed to be contiguous (e.g., could be 1, 3, 5).

So always use FIRST/NEXT when dealing with **associative arrays**.

Would you like me to demonstrate the same using engineer\_id as the key instead of index 1, 2, 3? That's a common real-world approach.

##### You said:

yes

###### ChatGPT said:

Perfect! Let’s now modify the logic to use **engineer\_id as the key** in the associative array. This way, you're mapping actual business keys (like employee IDs) instead of artificial index positions like 1, 2, 3.

## ✅ Goal:

**Map each engineer\_id to engineer\_name** and print them using an associative array.

## 🔧 Step-by-Step PL/SQL Code:

plsql

CopyEdit

DECLARE

-- Define an associative array: key is engineer\_id, value is engineer\_name

TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY NUMBER;

eng\_names eng\_map;

-- Temporary variables

v\_id engineers.engineer\_id%TYPE;

BEGIN

-- Populate the associative array with engineer\_id as key

FOR rec IN (SELECT engineer\_id, engineer\_name FROM engineers) LOOP

eng\_names(rec.engineer\_id) := rec.engineer\_name;

END LOOP;

-- Loop through all keys (engineer IDs)

v\_id := eng\_names.FIRST;

WHILE v\_id IS NOT NULL LOOP

DBMS\_OUTPUT.PUT\_LINE('Engineer ID: ' || v\_id || ' → Name: ' || eng\_names(v\_id));

v\_id := eng\_names.NEXT(v\_id);

END LOOP;

END;

## 🔍 Explanation:

| **Line** | **Purpose** |
| --- | --- |
| TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY NUMBER; | Create a key-value structure, key is engineer\_id |
| eng\_names(rec.engineer\_id) := rec.engineer\_name; | Populate array using actual IDs |
| FIRST / NEXT | Iterates over all actual keys (not just 1..N) |
| DBMS\_OUTPUT.PUT\_LINE(...) | Prints both ID and name dynamically |

### ✅ Example Output (Assume DB has 3 engineers):

yaml

CopyEdit

Engineer ID: 101 → Name: John Smith

Engineer ID: 102 → Name: Alka Joshi

Engineer ID: 103 → Name: Sunil Mehra

## 🧠 Why This Is Useful in Real Systems:

| **Real Use Case** | **How Associative Arrays Help** |
| --- | --- |
| Ticket routing | You can lookup engineer name by ID instantly in memory |
| Reports | Avoid repeated joins between tickets and engineers |
| Email alerts | Personalize messages like: "Hello John Smith, a new ticket is assigned..." |

Would you like me to wrap this in a **procedure** so learners can reuse it with different teams or departments?

## Step 5: Nested Table for Bulk Tickets

DECLARE  
 TYPE issue\_tab IS TABLE OF VARCHAR2(100);  
 issues issue\_tab := issue\_tab('Login Fail', 'Slow Internet');  
BEGIN  
 FOR i IN 1..issues.COUNT LOOP  
 INSERT INTO tickets VALUES (ticket\_seq.NEXTVAL, issues(i), 'M', 102, SYSDATE, 'OPEN');  
 END LOOP;  
END;

Explanation :

## 👨‍💼 Project Manager’s Real-World Perspective

### 🎯 **Why are we doing this?**

In the real-world Helpdesk application, multiple tickets might need to be inserted at once:

* After a **network outage**, many users might report issues like:
  + Login failure
  + Slow internet
  + Printer not connecting
* These are **common issues** often logged in **bulk** by an IT Admin.

Instead of inserting them one-by-one manually, we **store them in a PL/SQL collection** and insert them efficiently via a loop.

## 🧠 Technical PL/SQL Developer Explanation

### ✅ PL/SQL Block Recap:

plsql

CopyEdit

DECLARE

TYPE issue\_tab IS TABLE OF VARCHAR2(100);

issues issue\_tab := issue\_tab('Login Fail', 'Slow Internet');

BEGIN

FOR i IN 1..issues.COUNT LOOP

INSERT INTO tickets VALUES (

ticket\_seq.NEXTVAL, -- auto-incremented ticket ID

issues(i), -- issue description

'M', -- severity (Medium)

102, -- assigned engineer ID

SYSDATE, -- current timestamp

'OPEN' -- ticket status

);

END LOOP;

END;

## 🧩 Breakdown of Each Line

| **Line** | **Description** |
| --- | --- |
| TYPE issue\_tab IS TABLE OF VARCHAR2(100); | Declares a **nested table type** that can hold strings (ticket issues) |
| issues issue\_tab := issue\_tab('Login Fail', 'Slow Internet'); | Initializes the table with two issue descriptions |
| FOR i IN 1..issues.COUNT LOOP | Iterates over the nested table using its count |
| ticket\_seq.NEXTVAL | Auto-generates a unique ticket\_id using a sequence |
| INSERT INTO tickets ... | Inserts each issue as a new row in the tickets table |

## ✅ Output (Assuming sequence starts at 1001)

yaml

CopyEdit

Ticket 1001: Login Fail

Ticket 1002: Slow Internet

## 🔍 Why Use a **Nested Table** Here?

| **Feature** | **Benefit** |
| --- | --- |
| Dynamic size | You don’t need to know the number of issues beforehand |
| Table-like | You can treat it like a list or array and loop through it |
| Can be stored in DB | Unlike VARRAY, nested tables can be used in DB tables later if needed |
| Clean bulk insert logic | Simpler than writing separate insert statements for each ticket |

## ✅ Real Use Case

Let’s say:

* An engineer creates **5 tickets from a daily email report**.
* You store them in a nested table, and loop through to insert into the database.
* This avoids repeated INSERT blocks in the script and gives **scalable** and **clean code**.

## 🧪 Bonus Tip

You can even make this more dynamic:

plsql

CopyEdit

DECLARE

TYPE issue\_tab IS TABLE OF VARCHAR2(100);

issues issue\_tab := issue\_tab();

BEGIN

-- dynamically add issues

issues.EXTEND(2);

issues(1) := 'Printer not working';

issues(2) := 'Mouse replacement';

FOR i IN 1..issues.COUNT LOOP

INSERT INTO tickets VALUES (ticket\_seq.NEXTVAL, issues(i), 'L', 103, SYSDATE, 'OPEN');

END LOOP;

END;

Would you like this to be converted into a **procedure** where you can pass a list of issues dynamically (e.g., via a web app or form)?

##### You said:

  issues issue\_tab := issue\_tab('Login Fail', 'Slow Internet'); but how come we will take these values dynamically which will be inserted

###### ChatGPT said:

Excellent and practical question! 👏  
You're now thinking **like a backend engineer** working on real-time systems — let’s walk through this step by step.

## 🧠 Problem Statement:

The code currently hardcodes:

plsql

CopyEdit

issues issue\_tab := issue\_tab('Login Fail', 'Slow Internet');

But in real projects, issues must be **dynamic** — for example:

* Pulled from a **frontend form**
* Captured from a **text file or log**
* Read from **another database table**
* Retrieved via **user-defined inputs**

## ✅ Realistic Ways to Insert Dynamic Issue Data

Let’s explore 3 **production-level** ways you can dynamically populate issues collection:

### ✅ **1. From Another Table (Most Common in Real Systems)**

#### 🎯 Use Case:

Admin stores common issues in a table issue\_templates, and we load those dynamically.

#### 🔧 Setup:

sql

CopyEdit

CREATE TABLE issue\_templates (

issue\_text VARCHAR2(100)

);

-- Sample data

INSERT INTO issue\_templates VALUES ('Login Fail');

INSERT INTO issue\_templates VALUES ('VPN Timeout');

INSERT INTO issue\_templates VALUES ('Outlook Crash');

COMMIT;

#### 🔁 Updated Code Using BULK COLLECT:

plsql

CopyEdit

DECLARE

TYPE issue\_tab IS TABLE OF VARCHAR2(100);

issues issue\_tab;

BEGIN

-- Pull issue descriptions from issue\_templates table

SELECT issue\_text BULK COLLECT INTO issues FROM issue\_templates;

-- Insert into tickets

FOR i IN 1..issues.COUNT LOOP

INSERT INTO tickets VALUES (

ticket\_seq.NEXTVAL,

issues(i),

'M',

102,

SYSDATE,

'OPEN'

);

END LOOP;

END;

✅ Now the issues list is completely **dynamic**, fetched at runtime from a table.

### ✅ **2. From User Parameters (Stored Procedure)**

If you’re building an API or UI form that submits multiple issues:

plsql

CopyEdit

CREATE OR REPLACE PROCEDURE insert\_bulk\_issues (

p\_issue1 IN VARCHAR2,

p\_issue2 IN VARCHAR2

) AS

TYPE issue\_tab IS TABLE OF VARCHAR2(100);

issues issue\_tab := issue\_tab(p\_issue1, p\_issue2);

BEGIN

FOR i IN 1..issues.COUNT LOOP

INSERT INTO tickets VALUES (

ticket\_seq.NEXTVAL,

issues(i),

'L',

103,

SYSDATE,

'OPEN'

);

END LOOP;

END;

#### Call Example:

sql

CopyEdit

EXEC insert\_bulk\_issues('New User Request', 'Software Install');

✅ Good for interactive apps or forms where the user enters 2–5 issues.

### ✅ **3. From File or External Data (Advanced)**

Oracle allows external tables or UTL\_FILE package to **read from flat files** (CSV, logs, etc.), which can be **BULK COLLECTED** into the collection.

Let me know if you want an example using UTL\_FILE.

## ✅ Summary: How to Make It Dynamic

| **Method** | **When to Use** |
| --- | --- |
| BULK COLLECT FROM TABLE | Issues stored in DB already (admin interface, scheduled jobs) |
| VARRAY / Parameters | Issues coming from front-end app (API call, UI form) |
| UTL\_FILE | Bulk import from files (ticket migration, external reporting tools) |

Would you like to turn this into a **procedure that accepts a department ID** and automatically assigns issues to a selected engineer with logging?

##### You said:

yes

###### ChatGPT said:

Perfect! Let's build a **modular and dynamic procedure** that takes in:

* ✅ A **department ID**
* ✅ Automatically fetches related issues from an issue\_templates table
* ✅ Assigns them to an engineer in that department
* ✅ Inserts them into the tickets table
* ✅ Logs each action via DBMS\_OUTPUT

## ✅ Step-by-Step Solution

### 🏗 Prerequisites – Sample Tables

Ensure these exist:

sql

CopyEdit

-- Engineers Table

CREATE TABLE engineers (

engineer\_id NUMBER PRIMARY KEY,

engineer\_name VARCHAR2(100),

department VARCHAR2(50)

);

-- Tickets Table

CREATE TABLE tickets (

ticket\_id NUMBER PRIMARY KEY,

issue\_desc VARCHAR2(200),

severity CHAR(1),

assigned\_to NUMBER,

created\_on DATE,

status VARCHAR2(20)

);

-- Ticket Sequence

CREATE SEQUENCE ticket\_seq START WITH 1001 INCREMENT BY 1;

-- Issue Templates Table

CREATE TABLE issue\_templates (

department VARCHAR2(50),

issue\_text VARCHAR2(100)

);

### 🧪 Sample Data

sql

CopyEdit

-- Engineers

INSERT INTO engineers VALUES (101, 'Riya', 'IT');

INSERT INTO engineers VALUES (102, 'Amit', 'HR');

INSERT INTO engineers VALUES (103, 'Sara', 'IT');

-- Issue Templates

INSERT INTO issue\_templates VALUES ('IT', 'VPN Not Working');

INSERT INTO issue\_templates VALUES ('IT', 'Slow Internet');

INSERT INTO issue\_templates VALUES ('HR', 'Payslip Not Downloading');

INSERT INTO issue\_templates VALUES ('HR', 'Leave Balance Error');

COMMIT;

## 🧠 Final Stored Procedure: bulk\_issue\_create

plsql

CopyEdit

CREATE OR REPLACE PROCEDURE bulk\_issue\_create (

p\_dept IN VARCHAR2

) AS

-- Collection to hold dynamic issues

TYPE issue\_tab IS TABLE OF VARCHAR2(100);

issues issue\_tab;

-- Engineer ID to assign (get first one in that dept)

v\_engineer\_id engineers.engineer\_id%TYPE;

BEGIN

-- Get engineer for the given department

SELECT engineer\_id INTO v\_engineer\_id

FROM engineers

WHERE department = p\_dept

AND ROWNUM = 1;

-- Collect department-wise issue templates

SELECT issue\_text

BULK COLLECT INTO issues

FROM issue\_templates

WHERE department = p\_dept;

-- Insert each issue into tickets table

FOR i IN 1..issues.COUNT LOOP

INSERT INTO tickets VALUES (

ticket\_seq.NEXTVAL,

issues(i),

'M', -- default severity

v\_engineer\_id,

SYSDATE,

'OPEN'

);

DBMS\_OUTPUT.PUT\_LINE('Inserted: ' || issues(i) || ' → Engineer: ' || v\_engineer\_id);

END LOOP;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('No engineer or issues found for department: ' || p\_dept);

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

END;

## 🧪 Execute the Procedure

sql

CopyEdit

EXEC bulk\_issue\_create('IT');

EXEC bulk\_issue\_create('HR');

EXEC bulk\_issue\_create('Marketing'); -- should hit NO\_DATA\_FOUND

## ✅ Output Example:

yaml

CopyEdit

Inserted: VPN Not Working → Engineer: 101

Inserted: Slow Internet → Engineer: 101

Inserted: Payslip Not Downloading → Engineer: 102

Inserted: Leave Balance Error → Engineer: 102

No engineer or issues found for department: Marketing

## ✅ What You've Learned

| **Concept** | **Implemented** |
| --- | --- |
| **Dynamic collections** | Nested table for issues |
| **Parameterized procedure** | Accept p\_dept dynamically |
| **BULK COLLECT** | Pulls all department issues in one shot |
| **Auto assignment** | Finds the engineer dynamically |
| **Logging** | Done with DBMS\_OUTPUT |
| **Exception handling** | NO\_DATA\_FOUND + OTHERS |

## Step 6: Cursor with Record and PRAGMA Exception

DECLARE  
 CURSOR cur\_tickets IS SELECT ticket\_id, issue\_desc FROM tickets WHERE assigned\_to = 101;  
 r\_tickets cur\_tickets%ROWTYPE;  
 e\_no\_ticket EXCEPTION;  
 PRAGMA EXCEPTION\_INIT(e\_no\_ticket, -20001);  
 v\_count NUMBER := 0;  
BEGIN  
 OPEN cur\_tickets;  
 LOOP  
 FETCH cur\_tickets INTO r\_tickets;  
 EXIT WHEN cur\_tickets%NOTFOUND;  
 v\_count := v\_count + 1;  
 DBMS\_OUTPUT.PUT\_LINE('Ticket ID: ' || r\_tickets.ticket\_id);  
 END LOOP;  
 CLOSE cur\_tickets;  
  
 IF v\_count = 0 THEN  
 RAISE\_APPLICATION\_ERROR(-20001, 'No Tickets Found');  
 END IF;  
EXCEPTION  
 WHEN e\_no\_ticket THEN  
 DBMS\_OUTPUT.PUT\_LINE('Handled: ' || SQLERRM);  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Unexpected Error: ' || SQLERRM);  
END;

Explanation :

# ✅ **Step 6: Cursor with RECORD and PRAGMA Exception**

### 🧩 Goal:

We want to **retrieve all tickets assigned to a particular engineer (engineer\_id = 101)** and display them one by one. If no tickets are found, the system should **gracefully handle it using PRAGMA-defined exception logic**.

## 🔍 Business Use Case (Real Scenario):

Imagine you are in charge of an **IT Support dashboard**, and team leads want to see:

“All current tickets assigned to their engineers — and a clean message if none are found.”

But instead of just printing “0 rows found”, we want to:

* **Log each ticket individually**
* **Raise a meaningful error if nothing is assigned**
* Use a **controlled, project-specific error code** with **PRAGMA EXCEPTION\_INIT**

## 📦 Breakdown of Code

plsql

CopyEdit

DECLARE

* Start of an **anonymous PL/SQL block**.

### 1️⃣ **CURSOR Declaration**

plsql

CopyEdit

CURSOR cur\_tickets IS

SELECT ticket\_id, issue\_desc

FROM tickets

WHERE assigned\_to = 101;

* Declares an **explicit cursor** that fetches tickets assigned to engineer with ID = 101.
* ticket\_id and issue\_desc are selected for display.

🧠 Why cursor? Because we want to fetch **row-by-row**, print, and have control over the loop.

### 2️⃣ **Record Declaration**

plsql

CopyEdit

r\_tickets cur\_tickets%ROWTYPE;

* Declares a record variable that will hold **one row at a time** from the cursor.
* %ROWTYPE makes the record structure automatically match the cursor structure.

### 3️⃣ **Custom Exception Setup using PRAGMA**

plsql

CopyEdit

e\_no\_ticket EXCEPTION;

PRAGMA EXCEPTION\_INIT(e\_no\_ticket, -20001);

* e\_no\_ticket is a **user-defined exception**.
* PRAGMA EXCEPTION\_INIT links it to **Oracle error code -20001**, making it easier to raise or catch custom logic.

✅ Great practice when you're defining business-specific errors in enterprise systems.

### 4️⃣ **Ticket Loop Using Cursor**

plsql

CopyEdit

v\_count NUMBER := 0;

OPEN cur\_tickets;

LOOP

FETCH cur\_tickets INTO r\_tickets;

EXIT WHEN cur\_tickets%NOTFOUND;

v\_count := v\_count + 1;

DBMS\_OUTPUT.PUT\_LINE('Ticket ID: ' || r\_tickets.ticket\_id);

END LOOP;

CLOSE cur\_tickets;

* v\_count keeps track of how many rows we fetched.
* **Each row is printed** using DBMS\_OUTPUT.
* If nothing is fetched, v\_count remains 0.

🔄 This is a classic **explicit cursor loop**: open → fetch → exit when not found → close.

### 5️⃣ **Raise Application Error If No Tickets Found**

plsql

CopyEdit

IF v\_count = 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'No Tickets Found');

END IF;

* Raises a **custom error** linked to the PRAGMA exception, to simulate a failure and let higher apps know **no tickets were assigned**.

### 6️⃣ **Exception Handling Block**

plsql

CopyEdit

EXCEPTION

WHEN e\_no\_ticket THEN

DBMS\_OUTPUT.PUT\_LINE('Handled: ' || SQLERRM);

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Unexpected Error: ' || SQLERRM);

* WHEN e\_no\_ticket: catches your custom logic (no rows).
* WHEN OTHERS: a catch-all for any runtime errors like data type mismatch, etc.

## ✅ Summary: Why This is Best Practice

| **Purpose** | **How it's handled here** |
| --- | --- |
| Read multiple rows | Cursor + LOOP |
| Flexible row handling | %ROWTYPE automatically adapts to selected columns |
| Controlled failure handling | PRAGMA exception to raise meaningful custom errors |
| Graceful display | DBMS\_OUTPUT.PUT\_LINE to print ticket ID |

## 📊 Real-World Use Case in Companies:

| **Role** | **Use of This Block** |
| --- | --- |
| Team Lead | Check open tickets per engineer |
| Operations | Validate ticket assignment distribution |
| Dashboard Dev | Wrap logic inside procedure for UI-driven app |
| QA | Ensure zero-ticket cases are properly handled |

## 🧪 Want a version that takes engineer\_id as a parameter?

✅ Already added as a procedure in the earlier ticket\_pkg.

Let me know if you'd like to create a **function** version or use this logic in a **dashboard view** or **notification simulation table**.

## Step 7: Views and Inline Views for Dashboard

CREATE OR REPLACE VIEW ticket\_summary AS  
SELECT assigned\_to, COUNT(\*) AS total  
FROM tickets  
GROUP BY assigned\_to;  
  
-- Inline View  
SELECT \* FROM (  
 SELECT issue\_desc, created\_on FROM tickets ORDER BY created\_on DESC  
) WHERE ROWNUM <= 3;

Explanation:

## 🔹 What’s the Real-World Purpose?

Imagine you're managing a Helpdesk System. On your dashboard, you'd want:

* **How many tickets** are assigned to each engineer → summary view
* **Latest 3 raised tickets** → recent activity feed

This step allows us to build a **read-only reporting layer** for such insights using **Views and Inline Views**.

## 🔸 PART 1: ticket\_summary – A **Named View**

### 🔧 Code:

sql

CopyEdit

CREATE OR REPLACE VIEW ticket\_summary AS

SELECT assigned\_to, COUNT(\*) AS total

FROM tickets

GROUP BY assigned\_to;

### 🧠 What Is This?

A **View** is a **virtual table** based on a SQL query. You can query it just like a regular table.

This one shows:

* Who (assigned\_to) has tickets
* How many tickets (COUNT(\*)) they have

### ✅ Output Example:

| **assigned\_to** | **total** |
| --- | --- |
| 101 | 4 |
| 102 | 2 |

You can now run:

sql

CopyEdit

SELECT \* FROM ticket\_summary;

### 🧩 When Do We Use This?

* For **dashboards**, reports, or exporting data
* When business users need a simplified layer without querying raw tables
* Used often in **analytics** or **BI tools** connected to Oracle DB

## 🔸 PART 2: Inline View – **Recent Tickets Report**

### 🔧 Code:

sql

CopyEdit

SELECT \* FROM (

SELECT issue\_desc, created\_on FROM tickets

ORDER BY created\_on DESC

) WHERE ROWNUM <= 3;

### 🧠 What’s an Inline View?

* An **inline view** is a **subquery** in the FROM clause
* It’s temporary and unnamed — unlike the ticket\_summary view
* Used for **sorting, filtering, limiting** rows

### ✅ What This Does:

1. Sorts tickets by most recent created\_on
2. Picks **top 3 recent tickets** only using ROWNUM <= 3

### ✅ Output Example:

| **issue\_desc** | **created\_on** |
| --- | --- |
| VPN Not Working | 01-JUL-2025 09:00 |
| Login Fail | 01-JUL-2025 08:45 |
| Email Not Syncing | 01-JUL-2025 08:30 |

## 🔍 Comparison: View vs Inline View

| **Feature** | **View** | **Inline View** |
| --- | --- | --- |
| Is it saved? | ✅ Yes, created with CREATE VIEW | ❌ No, temporary |
| Reusable? | ✅ Can be reused in multiple queries | ❌ Only within the query |
| Name | ticket\_summary | None |
| Best for? | Dashboards, user access | Sorting, filtering, ranking |

## 👨‍💼 Project Manager Insight:

We don’t want our business analysts writing GROUP BY or complex joins. So we give them views like ticket\_summary to safely query pre-prepared logic.

Similarly, the dashboard needs to show the “3 most recent tickets.” An inline view ensures **performance and clarity** without needing to create extra views.

## 🧪 BONUS: Query Examples

**1. Using the View**:

sql

CopyEdit

SELECT \* FROM ticket\_summary WHERE total > 3;

**2. Filtering Inline View by Severity**:

sql

CopyEdit

SELECT \* FROM (

SELECT issue\_desc, severity, created\_on FROM tickets

ORDER BY created\_on DESC

) WHERE severity = 'H' AND ROWNUM <= 5;

Would you like to now create a **materialized view** (refreshed daily) or embed these into a **procedure that generates a report**?

##### You said:

yes

###### ChatGPT said:

Great! Let's build on the existing views by taking it to the **next enterprise-ready level** using:

## ✅ Goal:

Create a **Materialized View** and a **Report Procedure** that:

1. **Stores a snapshot** of the helpdesk ticket summary (materialized view)
2. **Generates a daily summary report** from that materialized view (PL/SQL procedure)
3. **Logs the report time and engineer ticket counts**

## 🔹 Part 1: Materialized View — mv\_ticket\_summary

### 🧠 What is a Materialized View?

* It **physically stores** the result of a query.
* Useful for **precomputed analytics** or **expensive joins/aggregates**.
* Can be refreshed **manually or periodically**.

### 🧱 Create the Materialized View:

sql

CopyEdit

CREATE MATERIALIZED VIEW mv\_ticket\_summary

BUILD IMMEDIATE

REFRESH ON DEMAND

AS

SELECT assigned\_to, COUNT(\*) AS total

FROM tickets

GROUP BY assigned\_to;

🔄 You can later **refresh** it by:

sql

CopyEdit

EXEC DBMS\_MVIEW.REFRESH('MV\_TICKET\_SUMMARY');

## 🔹 Part 2: Create Report Log Table (Optional)

We’ll store the timestamp and engineer's ticket count:

sql

CopyEdit

CREATE TABLE report\_log (

report\_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

engineer\_id NUMBER,

ticket\_count NUMBER,

generated\_on DATE DEFAULT SYSDATE

);

## 🔹 Part 3: Procedure to Generate Ticket Summary Report

### 🛠 generate\_ticket\_report – Fetch from materialized view and log

plsql

CopyEdit

CREATE OR REPLACE PROCEDURE generate\_ticket\_report IS

BEGIN

-- Refresh the MV first

DBMS\_MVIEW.REFRESH('MV\_TICKET\_SUMMARY');

-- Insert the summary into the report\_log

INSERT INTO report\_log (engineer\_id, ticket\_count)

SELECT assigned\_to, total

FROM mv\_ticket\_summary;

DBMS\_OUTPUT.PUT\_LINE('Ticket Summary Report Generated at ' || SYSDATE);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Report generation failed: ' || SQLERRM);

END;

## ✅ Run the Report Procedure

sql

CopyEdit

EXEC generate\_ticket\_report;

## 🧾 View the Report Log

sql

CopyEdit

SELECT \* FROM report\_log ORDER BY generated\_on DESC;

## 🧠 Real-World Use Case:

| **Element** | **Purpose** |
| --- | --- |
| Materialized View | Precomputed daily ticket counts (engineer-level) |
| Procedure | Logs each report generation (when, what data) |
| DBMS\_OUTPUT | Console trace of job status |
| REFRESH ON DEMAND | Manual refresh — can later schedule with a job |

## 💡 Optional Enhancements:

* ⏱ Schedule using DBMS\_SCHEDULER for **daily refresh**
* 📥 Add an email\_report(to\_address) procedure to send the output
* 📊 Create a **dashboard** reading from mv\_ticket\_summary and report\_log

Would you like help scheduling this procedure automatically via DBMS\_SCHEDULER or creating a version that also sends an email report?

##### You said:

why we are using materialized it is autorefreshing we are calling it manually and the normal procedure also work like this we call it to execute it

###### ChatGPT said:

That's an **excellent question** — and one that comes up often when choosing between **Views** vs **Materialized Views** vs **Procedures** in real-world enterprise environments like Oracle.

Let me break this down **step by step**, as a **Project Manager explaining to both developers and analysts**:

## 🧩 Why Use a **Materialized View** Instead of Just a Regular View or Procedure?

| **Aspect** | **Regular View** | **Materialized View** | **Procedure (without MV)** |
| --- | --- | --- | --- |
| **Storage** | Virtual (no data stored) | Physically stores data in DB | No storage unless inserted manually |
| **Performance** | Computed every time it's queried | Fast – data is precomputed | Fast, but only when data is fetched/inserted |
| **Usage in Reports** | Good for real-time dashboards | Good for scheduled reports, BI dashboards | Best for custom logic or workflows |
| **Refresh Control** | Always up-to-date (but slow for big data) | Can be manual, on-commit, or scheduled | Needs full logic and execution |
| **Join & Aggregation Cost** | Every query incurs cost | Done once at refresh | Depends on logic |
| **Ease of Use for Analysts** | Requires complex SQL | Easy SELECT \* from MV | Harder unless pre-formatted |

### 🔸 So Why Use a Materialized View in Our Project?

In our IT Helpdesk System:

| **Goal** | **Reason for Materialized View** |
| --- | --- |
| Dashboard needs ticket counts | Instead of running GROUP BY each time (slow), we use MV |
| Used daily by many stakeholders | MVs reduce **read pressure** on large tickets table |
| We want to refresh periodically | MV allows **manual or scheduled refresh** as needed |
| Plug into BI / Power BI | MVs work well with **BI tools** that expect static data |

## ✅ Why Refresh Manually Then?

You're right — in our example we used:

sql

CopyEdit

REFRESH ON DEMAND

Which means we **control when to refresh**.

This is useful because:

* ✅ You may **not want to refresh** with every change (performance hit)
* ✅ You might tie refresh to a **daily schedule** (e.g., midnight jobs)
* ✅ You may want to refresh **after new ticket inserts** only

🧠 So **manual refresh** is not a drawback — it’s a **design decision** to avoid unnecessary load.

If needed, you can also change it to:

sql

CopyEdit

REFRESH ON COMMIT

⚠️ But this causes the MV to refresh **every time the base table changes**, which may not be desirable if the ticket table is frequently written to.

## 💡 Comparison Summary (Use-Case Based):

| **Use Case** | **Best Option** |
| --- | --- |
| Real-time report with current data | View |
| Daily dashboard that reads thousands of rows | Materialized View |
| Ticket creation logic, custom alerts | Procedure |
| Export static snapshot for BI | Materialized View |
| Ad-hoc on-demand analytics | View or Procedure |

### 🔚 Final Thought

**Materialized View** gives us a “readymade snapshot,” which is **ideal for reporting, analytics, and dashboards**, especially when multiple teams or tools access the database.

Would you like me to demonstrate how to **auto-refresh the MV daily** using DBMS\_SCHEDULER, or would you like to explore **View + Procedure** alternatives further?

# Real-World Concepts Mapped (Case Study 1)

| Topic | Implemented in |
| --- | --- |
| Variables & Anchored | Ticket creation |
| RECORD & OBJECT | ticket\_obj, %ROWTYPE |
| Collections | eng\_map, issue\_tab, ticket\_list |
| Exception Handling | PRAGMA, user-defined |
| Control Structures | CASE, FOR, IF |
| Cursors | Explicit, %ROWTYPE, parameterized |
| Views | Summary view, inline view |

# 

# 📘 Case Study 2: Student Performance & Evaluation System (Fully Enhanced)

## Objective:

Automate student grading, classification, top scorer listing, and performance summaries using complete PL/SQL constructs.

## Components Used:

* ✅ Scalar & Anchored Variables
* ✅ RECORD, OBJECT TYPE
* ✅ Collections: Associative Array, VARRAY, Nested Table
* ✅ Views, Inline Views
* ✅ Exception Handling: PRAGMA, User-defined, Predefined
* ✅ Cursors: Implicit, Explicit, Parameterized
* ✅ Control Structures: IF-ELSE, CASE, FOR, WHILE

## Step 1: Create Base Table and Sequence

CREATE TABLE students (  
 student\_id NUMBER PRIMARY KEY,  
 student\_name VARCHAR2(100),  
 marks NUMBER,  
 dept VARCHAR2(50)  
);  
  
CREATE SEQUENCE stud\_seq START WITH 1;

## Step 2: Insert Data Using Variables and Anchored Type

DECLARE  
 v\_id NUMBER := stud\_seq.NEXTVAL;  
 v\_name students.student\_name%TYPE := 'Ravi';  
 v\_marks students.marks%TYPE := 85;  
 v\_dept students.dept%TYPE := 'CSE';  
BEGIN  
 INSERT INTO students VALUES (v\_id, v\_name, v\_marks, v\_dept);  
END;

## Step 3: Grading with RECORD, IF-ELSE, CASE

DECLARE  
 TYPE stud\_rec IS RECORD (  
 name VARCHAR2(100),  
 marks NUMBER  
 );  
 r stud\_rec;  
 v\_grade CHAR(1);  
BEGIN  
 SELECT student\_name, marks INTO r FROM students WHERE student\_id = 1;  
 IF r.marks >= 90 THEN v\_grade := 'A';  
 ELSIF r.marks >= 80 THEN v\_grade := 'B';  
 ELSIF r.marks >= 70 THEN v\_grade := 'C';  
 ELSE v\_grade := 'F';  
 END IF;  
  
 CASE v\_grade  
 WHEN 'A' THEN DBMS\_OUTPUT.PUT\_LINE('Excellent');  
 WHEN 'B' THEN DBMS\_OUTPUT.PUT\_LINE('Good');  
 ELSE DBMS\_OUTPUT.PUT\_LINE('Needs Improvement');  
 END CASE;  
END;

## Step 4: Top Scorers with VARRAY and Object

CREATE OR REPLACE TYPE stud\_obj AS OBJECT (  
 id NUMBER,  
 name VARCHAR2(100),  
 marks NUMBER  
);  
  
CREATE OR REPLACE TYPE topper\_array IS VARRAY(3) OF stud\_obj;  
  
DECLARE  
 v\_topper topper\_array := topper\_array();  
BEGIN  
 FOR r IN (  
 SELECT student\_id, student\_name, marks FROM students ORDER BY marks DESC FETCH FIRST 3 ROWS ONLY  
 ) LOOP  
 v\_topper.EXTEND;  
 v\_topper(v\_topper.LAST) := stud\_obj(r.student\_id, r.student\_name, r.marks);  
 END LOOP;  
  
 FOR i IN 1..v\_topper.COUNT LOOP  
 DBMS\_OUTPUT.PUT\_LINE('Topper #' || i || ': ' || v\_topper(i).name);  
 END LOOP;  
END;

## Step 5: Associative Array for Department-Wise Budget (Simulated)

DECLARE  
 TYPE dept\_budget\_type IS TABLE OF NUMBER INDEX BY VARCHAR2(10);  
 dept\_budget dept\_budget\_type;  
BEGIN  
 dept\_budget('CSE') := 100000;  
 dept\_budget('ECE') := 80000;  
 DBMS\_OUTPUT.PUT\_LINE('CSE Budget: ' || dept\_budget('CSE'));  
END;

## Step 6: Nested Table to Collect Marks Dynamically

DECLARE  
 TYPE marks\_table IS TABLE OF NUMBER;  
 v\_marks marks\_table;  
BEGIN  
 SELECT marks BULK COLLECT INTO v\_marks FROM students;  
 FOR i IN 1..v\_marks.COUNT LOOP  
 DBMS\_OUTPUT.PUT\_LINE('Marks: ' || v\_marks(i));  
 END LOOP;  
END;

## Step 7: Cursor with PRAGMA and User-Defined Exception

DECLARE  
 CURSOR cur\_dept(p\_dept VARCHAR2) IS  
 SELECT student\_name FROM students WHERE dept = p\_dept;  
 e\_no\_data\_found EXCEPTION;  
 PRAGMA EXCEPTION\_INIT(e\_no\_data\_found, -20011);  
 v\_found BOOLEAN := FALSE;  
BEGIN  
 FOR r IN cur\_dept('CSE') LOOP  
 v\_found := TRUE;  
 DBMS\_OUTPUT.PUT\_LINE('Student: ' || r.student\_name);  
 END LOOP;  
  
 IF NOT v\_found THEN  
 RAISE\_APPLICATION\_ERROR(-20011, 'No students found in CSE');  
 END IF;  
EXCEPTION  
 WHEN e\_no\_data\_found THEN  
 DBMS\_OUTPUT.PUT\_LINE('Handled: ' || SQLERRM);  
END;

## Step 8: View and Inline View for Summary

CREATE OR REPLACE VIEW dept\_summary AS  
SELECT dept, COUNT(\*) AS total FROM students GROUP BY dept;  
  
-- Inline View: Top 2 Students  
SELECT \* FROM (  
 SELECT student\_name, marks FROM students ORDER BY marks DESC  
) WHERE ROWNUM <= 2;

# ✅ Real-World Concepts Mapped (Case Study 2)

| Topic | Implemented in |
| --- | --- |
| Variables & Anchored | Student table inserts |
| RECORD & OBJECT | r record, stud\_obj |
| Collections | topper\_array, associative array, nested table |
| Exception Handling | PRAGMA, user-defined |
| Control Structures | IF, CASE, FOR |
| Cursors | Parameterized cursor |
| Views | Summary and inline |

Let me know if you want .sql versions, trainer guides, or practice questions for learners!