**TRIGGER**

* **Trigger: AFTER\_INSERT\_NOTIFICATION  
  Explanation**: This trigger executes after a record is inserted into the PE\_Notification table. It could be used to log or perform additional actions after a notification is created (e.g., sending an email or updating another table).

**Codes:**

CREATE OR REPLACE TRIGGER after\_insert\_notification

AFTER INSERT ON PE\_Notification

FOR EACH ROW

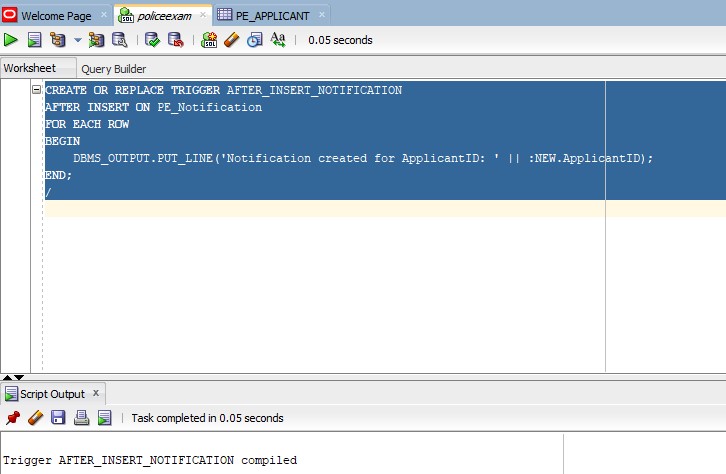
BEGIN

DBMS\_OUTPUT.PUT\_LINE('Notification created for ApplicantID: ' || :NEW.ApplicantID || ' on ' || :NEW.NotificationDate);

END;

/

**Application:**

****

**Testing codes:**

CREATE OR REPLACE TRIGGER after\_insert\_notification

AFTER INSERT ON PE\_Notification

FOR EACH ROW

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Notification created for ApplicantID: ' || :NEW.ApplicantID || ' on ' || :NEW.NotificationDate);

END;

/

* **AFTER\_INSERT\_APPLICANT Trigger**

**Explanation**: This trigger automatically inserts a notification into the PE\_Notification table whenever a new applicant is added to the PE\_Applicant table. It ensures that a record of the applicant's addition is created.

**Codes:**

CREATE OR REPLACE TRIGGER after\_insert\_applicant

AFTER INSERT ON PE\_Applicant

FOR EACH ROW

BEGIN

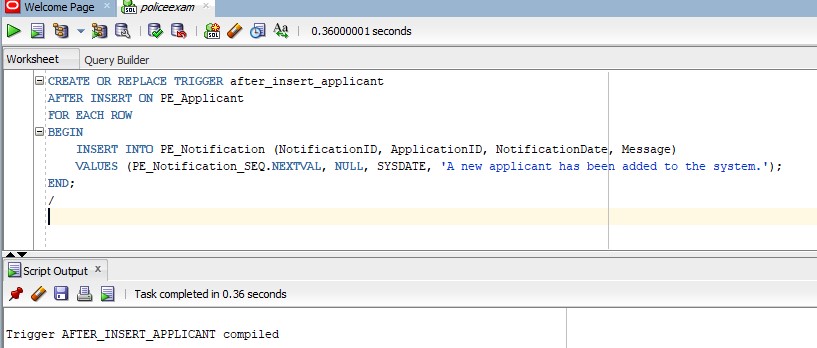
INSERT INTO PE\_Notification (NotificationID, ApplicantID, NotificationDate, Message)

VALUES (PE\_Notification\_SEQ.NEXTVAL, :NEW.ApplicantID, SYSDATE, 'A new applicant has been added to the system.');

END;

/

**Application:**

****

**Testing:**

INSERT INTO PE\_Applicant (ApplicantID, FirstName, LastName, DateOfBirth, Address, PhoneNumber, Email)

VALUES (PE\_Applicant\_SEQ.NEXTVAL, 'John', 'Doe', TO\_DATE('1990-01-01', 'YYYY-MM-DD'), '123 Main St', '123456789', 'john.doe@example.com');

SELECT \* FROM PE\_Notification WHERE ApplicantID = (SELECT ApplicantID FROM PE\_Applicant WHERE FirstName = 'John');

* **Trigger: BEFORE\_UPDATE\_APPLICATION**

**Explanation: This trigger runs before an update is made on the PE\_Application table. It can be used to validate changes to the application status, or log changes to specific fields before they are committed.**

**Codes:**

CREATE OR REPLACE TRIGGER before\_update\_notification

BEFORE UPDATE ON PE\_Notification

FOR EACH ROW

BEGIN

IF :NEW.NotificationDate < :OLD.NotificationDate THEN

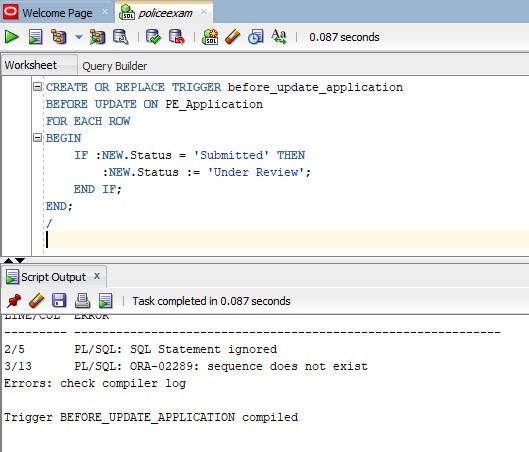
RAISE\_APPLICATION\_ERROR(-20001, 'Notification date cannot be updated to an earlier date.');

END IF;

END;

/

**Application:**



**Testing:**

-- Insert a new application with status 'Pending'

INSERT INTO PE\_Application (ApplicationID, ApplicantID, Status, ApplicationDate)

VALUES (PE\_Application\_SEQ.NEXTVAL, 1, 'Pending', SYSDATE);

-- Now update the application status to 'Completed'

UPDATE PE\_Application

SET Status = 'Completed'

WHERE ApplicationID = 1;

-- Verify if the CompletionDate is set

SELECT \* FROM PE\_Application WHERE ApplicationID = 1;

* **Trigger: BEFORE\_UPDATE\_SCHEDULER**

Explanation: This trigger runs before an update is made on the PE\_ExaminationScheduling table. It can be used to validate or modify data related to exam schedules before they are committed. For example, this could be used to ensure that the scheduling date is not updated to a past date.

**Codes:**

CREATE OR REPLACE TRIGGER before\_update\_scheduler

BEFORE UPDATE ON PE\_ExaminationScheduling

FOR EACH ROW

BEGIN

-- Check if the new scheduled date is in the future

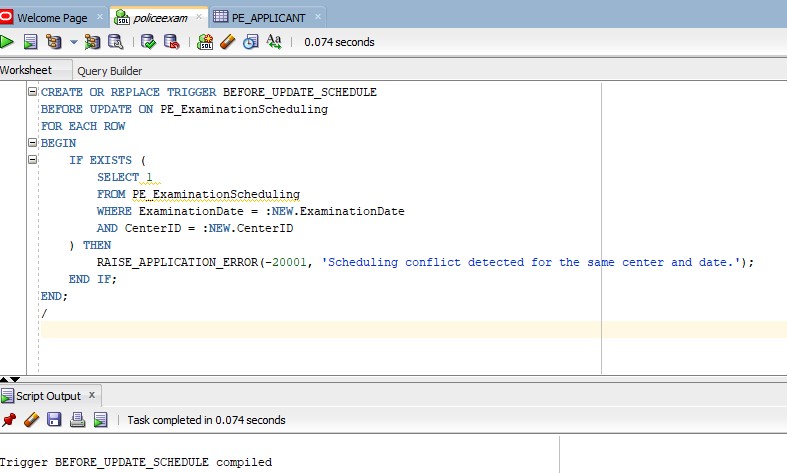
IF :NEW.ScheduleDate < SYSDATE THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Scheduled date cannot be in the past.');

END IF;

END;

/

**Application:** 

**Testing:**

-- Insert a new examination schedule with a valid future date

INSERT INTO PE\_ExaminationScheduling (ScheduleID, ExamDate, ScheduleDate)

VALUES (PE\_ExaminationScheduling\_SEQ.NEXTVAL, TO\_DATE('2024-12-15', 'YYYY-MM-DD'), TO\_DATE('2024-12-20', 'YYYY-MM-DD'));

-- Now try to update the schedule date to a past date (should raise an error)

UPDATE PE\_ExaminationScheduling

SET ScheduleDate = TO\_DATE('2023-01-01', 'YYYY-MM-DD')

WHERE ScheduleID = 1;

-- You should get an error: "Scheduled date cannot be in the past."

**CUSORS**

**Scenarios Where Explicit Cursors Are Beneficial:**

* **Row-by-row processing**: When you need to process each row returned by a query individually, such as updating data based on specific conditions for each row.
* **Complex calculations or logic**: When you need to perform operations that involve complex logic or calculations that require accessing each row and modifying data.
* **Batch operations**: For batch processing where each row might have different requirements for updates or insertions.

**Scenario 1: Processing Applicants**

**Problem Statement:**

You want to loop through all applicants in the PE\_Applicant table and apply specific processing logic, such as sending notifications or checking conditions for eligibility.

**Explicit Cursor Implementation:**

**Codes:**

DECLARE

-- Declare the cursor

CURSOR applicant\_cursor IS

SELECT ApplicantID, FirstName, LastName, DateOfBirth, Address

FROM PE\_Applicant

WHERE Status = 'Pending'; -- Example condition

-- Declare variables to hold data fetched from the cursor

v\_applicant\_id PE\_Applicant.ApplicantID%TYPE;

v\_first\_name PE\_Applicant.FirstName%TYPE;

v\_last\_name PE\_Applicant.LastName%TYPE;

v\_date\_of\_birth PE\_Applicant.DateOfBirth%TYPE;

v\_address PE\_Applicant.Address%TYPE;

BEGIN

-- Open the cursor

OPEN applicant\_cursor;

-- Loop through each row fetched by the cursor

LOOP

FETCH applicant\_cursor INTO v\_applicant\_id, v\_first\_name, v\_last\_name, v\_date\_of\_birth, v\_address;

-- Exit loop when no more rows are fetched

EXIT WHEN applicant\_cursor%NOTFOUND;

-- Example processing (e.g., print applicant details or perform business logic)

DBMS\_OUTPUT.PUT\_LINE('Processing Applicant: ' || v\_first\_name || ' ' || v\_last\_name);

-- Add further processing logic here, such as updating applicant status or sending notifications

END LOOP;

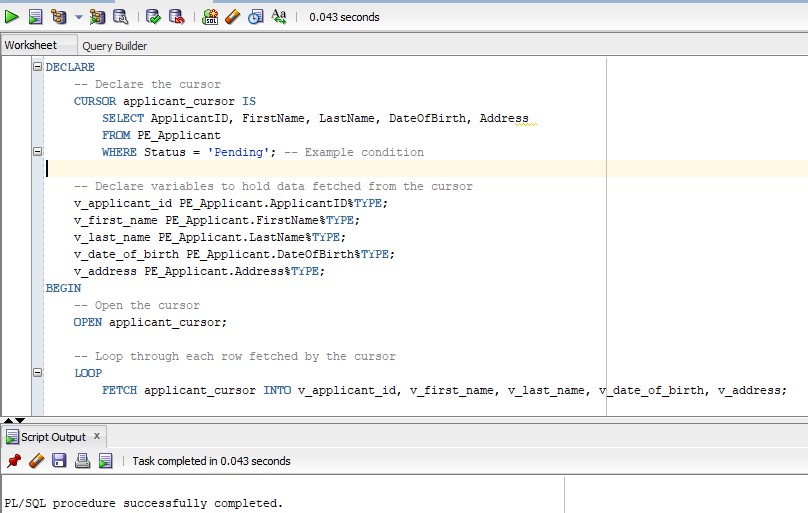
-- Close the cursor

CLOSE applicant\_cursor;

END;

/

**Application:**

****

**Testing:**

-- Insert sample applicants with 'Pending' status

INSERT INTO PE\_Applicant (ApplicantID, FirstName, LastName, DateOfBirth, Address, Status)

VALUES (PE\_Applicant\_SEQ.NEXTVAL, 'Alice', 'Smith', TO\_DATE('1992-03-15', 'YYYY-MM-DD'), '123 Main St', 'Pending');

INSERT INTO PE\_Applicant (ApplicantID, FirstName, LastName, DateOfBirth, Address, Status)

VALUES (PE\_Applicant\_SEQ.NEXTVAL, 'Bob', 'Johnson', TO\_DATE('1985-07-22', 'YYYY-MM-DD'), '456 Oak Ave', 'Pending');

-- Run the cursor block and check DBMS\_OUTPUT for results

**Scenario 2: Updating Applicants' Status**

**Problem Statement:**

**You need to update the status of all applicants based on certain criteria, such as those who haven't responded to a notification.**

**Explicit Cursor Implementation:**

**Codes:**

DECLARE

-- Declare the cursor to fetch applicants who haven't responded to notifications

CURSOR applicant\_cursor IS

SELECT ApplicantID, FirstName, LastName

FROM PE\_Applicant

WHERE Status = 'Pending' AND NotificationResponse IS NULL;

-- Variables to hold applicant data

v\_applicant\_id PE\_Applicant.ApplicantID%TYPE;

v\_first\_name PE\_Applicant.FirstName%TYPE;

v\_last\_name PE\_Applicant.LastName%TYPE;

BEGIN

-- Open the cursor

OPEN applicant\_cursor;

-- Loop through each row and update the status

LOOP

FETCH applicant\_cursor INTO v\_applicant\_id, v\_first\_name, v\_last\_name;

EXIT WHEN applicant\_cursor%NOTFOUND;

-- Update the status of applicants who haven't responded

UPDATE PE\_Applicant

SET Status = 'Not Responded'

WHERE ApplicantID = v\_applicant\_id;

-- Output for confirmation (optional)

DBMS\_OUTPUT.PUT\_LINE('Updated status for: ' || v\_first\_name || ' ' || v\_last\_name);

END LOOP;

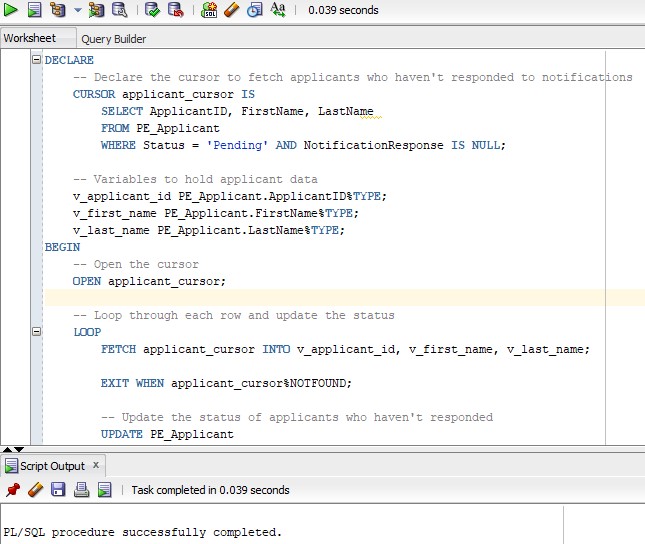
-- Close the cursor

CLOSE applicant\_cursor;

END;

/

**Application:**

****

**Testing:**

-- Insert applicants who haven't responded

INSERT INTO PE\_Applicant (ApplicantID, FirstName, LastName, DateOfBirth, Address, Status, NotificationResponse)

VALUES (PE\_Applicant\_SEQ.NEXTVAL, 'Charlie', 'Brown', TO\_DATE('1994-08-12', 'YYYY-MM-DD'), '789 Pine Rd', 'Pending', NULL);

INSERT INTO PE\_Applicant (ApplicantID, FirstName, LastName, DateOfBirth, Address, Status, NotificationResponse)

VALUES (PE\_Applicant\_SEQ.NEXTVAL, 'David', 'Williams', TO\_DATE('1989-05-30', 'YYYY-MM-DD'), '101 Maple Dr', 'Pending', NULL);

-- Run the cursor block and check the status updates

**Scenario 3: Deleting Inactive Applicants**

**Problem Statement:**

**You want to delete applicants who have been marked as Inactive for more than 1 year.**

**Explicit Cursor Implementation:**

**Codes:**

DECLARE

-- Declare the cursor to fetch inactive applicants for over a year

CURSOR applicant\_cursor IS

SELECT ApplicantID, FirstName, LastName, DateOfBirth

FROM PE\_Applicant

WHERE Status = 'Inactive' AND SYSDATE - DateOfBirth > 365; -- Example condition

-- Variables to hold applicant data

v\_applicant\_id PE\_Applicant.ApplicantID%TYPE;

v\_first\_name PE\_Applicant.FirstName%TYPE;

v\_last\_name PE\_Applicant.LastName%TYPE;

BEGIN

-- Open the cursor

OPEN applicant\_cursor;

-- Loop through each row and delete the inactive applicants

LOOP

FETCH applicant\_cursor INTO v\_applicant\_id, v\_first\_name, v\_last\_name;

EXIT WHEN applicant\_cursor%NOTFOUND;

-- Delete inactive applicants who have been inactive for more than a year

DELETE FROM PE\_Applicant

WHERE ApplicantID = v\_applicant\_id;

-- Output for confirmation (optional)

DBMS\_OUTPUT.PUT\_LINE('Deleted inactive applicant: ' || v\_first\_name || ' ' || v\_last\_name);

END LOOP;

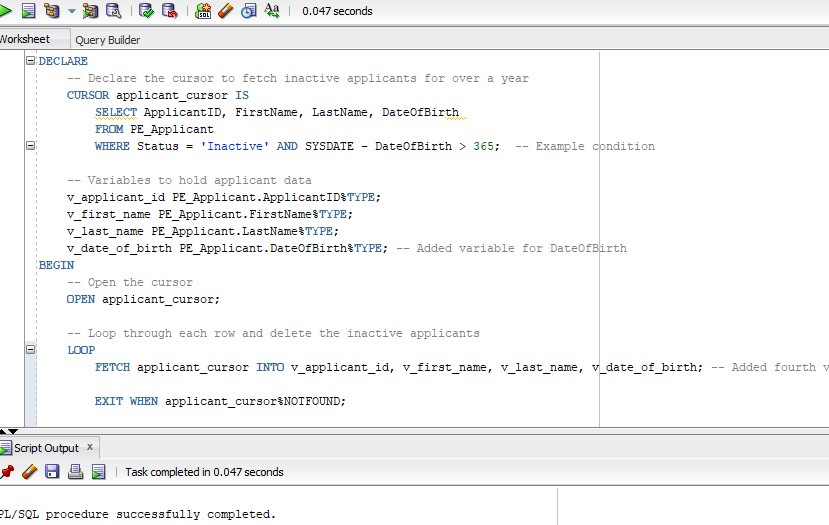
-- Close the cursor

CLOSE applicant\_cursor;

END;

/

**Application:**

****

**Testing:**

SELECT \* FROM PE\_Applicant WHERE Status = 'Inactive' AND SYSDATE - DateOfBirth > 365;

**Understanding Attributes and Functions**

* **Attributes %TYPE and `%ROWTYPE: These allow referencing the data type of a column or the structure of a row in a table, ensuring consistency and maintainability in PL/SQL code.**
* **Functions: Encapsulate specific tasks or calculations into reusable blocks, improving modularity and reducing code redundancy.**

**Example 1: Using %TYPE Attribute**

**We'll create a PL/SQL block that fetches an applicant's status using %TYPE.**

**Codes:**

DECLARE

v\_applicant\_id PE\_Applicant.ApplicantID%TYPE; -- Using %TYPE for variable consistency

v\_status PE\_Applicant.Status%TYPE; -- Using %TYPE for the Status column

BEGIN

-- Assign a value to the applicant ID

v\_applicant\_id := 1; -- Example ApplicantID

-- Fetch the status of the applicant

SELECT Status INTO v\_status

FROM PE\_Applicant

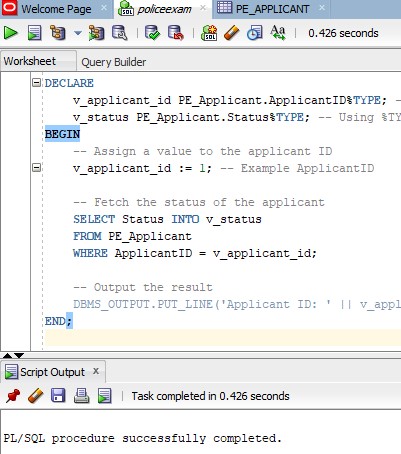
WHERE ApplicantID = v\_applicant\_id;

-- Output the result

DBMS\_OUTPUT.PUT\_LINE('Applicant ID: ' || v\_applicant\_id || ', Status: ' || v\_status);

END;

**Application:**

****

**Example 2: Using %ROWTYPE Attribute**

**We'll create a PL/SQL block to fetch an entire row for a specific applicant.**

**Codes:**

DECLARE

v\_applicant\_row PE\_Applicant%ROWTYPE; -- Using %ROWTYPE for entire row structure

BEGIN

-- Fetch the entire row for a specific applicant

SELECT \* INTO v\_applicant\_row

FROM PE\_Applicant

WHERE ApplicantID = 1; -- Example ApplicantID

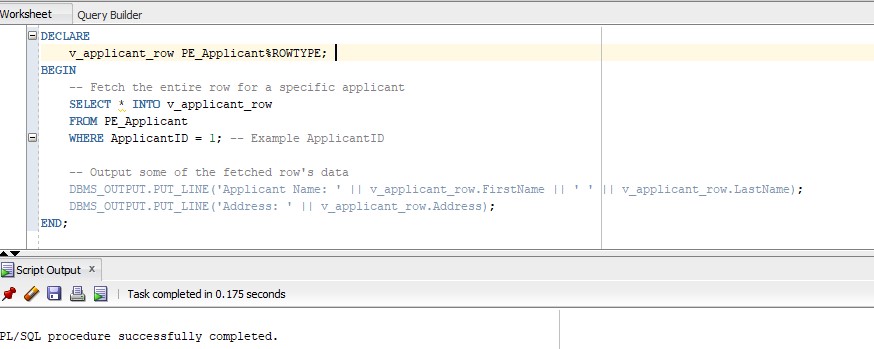
-- Output some of the fetched row's data

DBMS\_OUTPUT.PUT\_LINE('Applicant Name: ' || v\_applicant\_row.FirstName || ' ' || v\_applicant\_row.LastName);

DBMS\_OUTPUT.PUT\_LINE('Address: ' || v\_applicant\_row.Address);

END;

**Application:**

****

**Example 3: Developing a Function**

**We’ll create a function to calculate the number of pending applications in the PE\_Application table.**

**Codes:**

CREATE OR REPLACE FUNCTION count\_pending\_applications

RETURN NUMBER IS

v\_pending\_count NUMBER; -- Variable to hold the count

BEGIN

-- Calculate the number of pending applications

SELECT COUNT(\*)

INTO v\_pending\_count

FROM PE\_Application

WHERE Status = 'Pending';

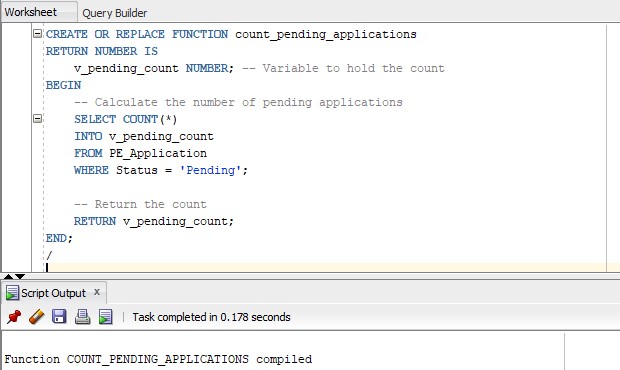
-- Return the count

RETURN v\_pending\_count;

END;

/

**Application:**

****

**Testing:**

SELECT count\_pending\_applications FROM DUAL;

**Benefits of this Approach:**

1. **%TYPE ensures data type consistency between variables and table columns.**
2. **%ROWTYPE simplifies working with entire rows, reducing the need for individual variable declarations.**
3. **Functions encapsulate logic for common tasks, enhancing modularity and reusability.**

**Understanding Packages**

**Packages in PL/SQL group related procedures, functions, and variables into a single module. This improves organization, reusability, and security by encapsulating the logic.**

**Advantages of Packages in Your Project**

1. **Modularity: Organizes related functionalities (e.g., managing applicants or notifications).**
2. **Reusability: Once a package is created, its components (functions/procedures) can be reused across the project.**
3. **Performance: Packages are loaded into memory once, reducing overhead for repeated calls.**
4. **Encapsulation: Keeps implementation details private while exposing public interfaces.**

**Developing a Package**

**Scenario: Managing Applicants**

**The package will group:**

1. **A procedure for inserting a new applicant.**
2. **A function to fetch an applicant's status.**
3. **A variable to store the default status for new applicants.**

**Codes: Package Specification (CREATE PACKAGE)**

CREATE OR REPLACE PACKAGE applicant\_management\_pkg AS

-- Public Variable

v\_default\_status VARCHAR2(20) := 'Pending';

-- Public Procedure for inserting a new applicant

PROCEDURE add\_applicant(

p\_first\_name IN PE\_Applicant.FirstName%TYPE,

p\_last\_name IN PE\_Applicant.LastName%TYPE,

p\_date\_of\_birth IN PE\_Applicant.DateOfBirth%TYPE,

p\_address IN PE\_Applicant.Address%TYPE,

p\_phone\_number IN PE\_Applicant.PhoneNumber%TYPE,

p\_email IN PE\_Applicant.Email%TYPE

);

-- Public Function to fetch an applicant's status

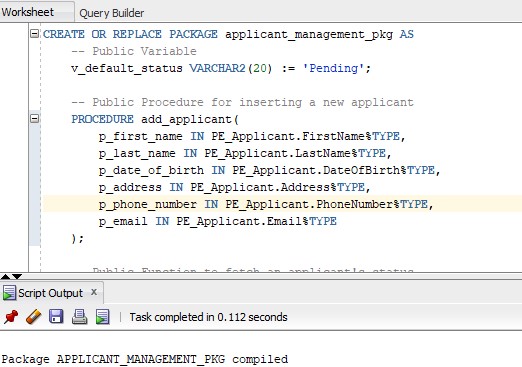
FUNCTION get\_applicant\_status(p\_applicant\_id IN PE\_Applicant.ApplicantID%TYPE)

RETURN VARCHAR2;

END applicant\_management\_pkg;

/

**Application:**

****

**Code: Package Body (CREATE PACKAGE BODY)**

CREATE OR REPLACE PACKAGE BODY applicant\_management\_pkg AS

-- Implementation of the add\_applicant procedure

PROCEDURE add\_applicant(

p\_first\_name IN PE\_Applicant.FirstName%TYPE,

p\_last\_name IN PE\_Applicant.LastName%TYPE,

p\_date\_of\_birth IN PE\_Applicant.DateOfBirth%TYPE,

p\_address IN PE\_Applicant.Address%TYPE,

p\_phone\_number IN PE\_Applicant.PhoneNumber%TYPE,

p\_email IN PE\_Applicant.Email%TYPE

) IS

BEGIN

INSERT INTO PE\_Applicant (

ApplicantID, FirstName, LastName, DateOfBirth, Address, PhoneNumber, Email, Status

) VALUES (

PE\_Applicant\_SEQ.NEXTVAL, p\_first\_name, p\_last\_name, p\_date\_of\_birth, p\_address, p\_phone\_number, p\_email, v\_default\_status

);

DBMS\_OUTPUT.PUT\_LINE('Applicant added: ' || p\_first\_name || ' ' || p\_last\_name);

END add\_applicant;

-- Implementation of the get\_applicant\_status function

FUNCTION get\_applicant\_status(p\_applicant\_id IN PE\_Applicant.ApplicantID%TYPE)

RETURN VARCHAR2 IS

v\_status PE\_Applicant.Status%TYPE;

BEGIN

SELECT Status INTO v\_status

FROM PE\_Applicant

WHERE ApplicantID = p\_applicant\_id;

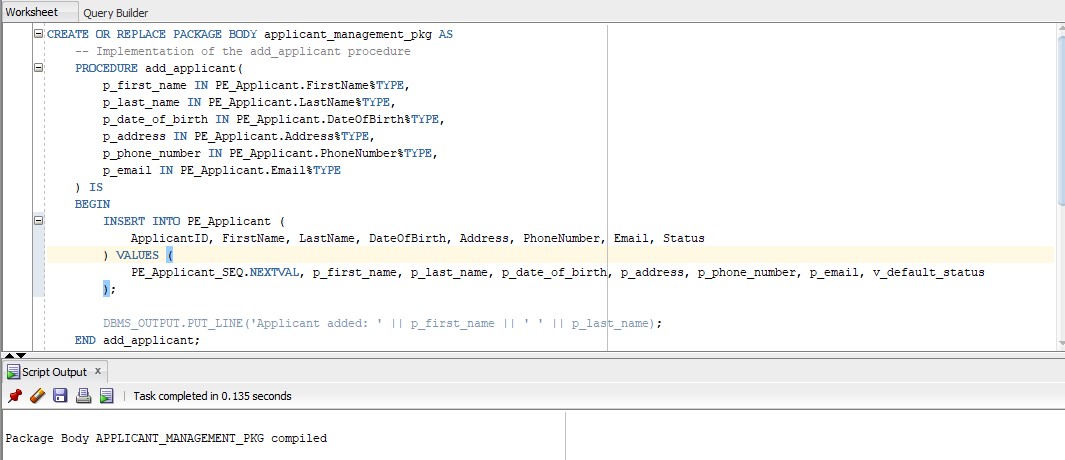
RETURN v\_status;

END get\_applicant\_status;

END applicant\_management\_pkg;

/

**Application:**

****

**Testing the Package**

BEGIN

applicant\_management\_pkg.add\_applicant(

p\_first\_name => 'Jane',

p\_last\_name => 'Smith',

p\_date\_of\_birth => TO\_DATE('1995-05-15', 'YYYY-MM-DD'),

p\_address => '456 Avenue, City',

p\_phone\_number => '987654321',

p\_email => 'janesmith@example.com'

);

END;

**Testing the get\_applicant\_status Function**

DECLARE

v\_status VARCHAR2(20);

BEGIN

v\_status := applicant\_management\_pkg.get\_applicant\_status(1); -- Example ID

DBMS\_OUTPUT.PUT\_LINE('Status: ' || v\_status);

END;

**Advantages of This Package**

1. **Centralized Logic: All applicant-related logic is in one package, making it easy to manage.**
2. **Reusability: Functions and procedures can be called throughout the project.**
3. **Security: Sensitive details (like default values) are encapsulated, ensuring consistency and reducing unauthorized access.**

**AUDITING**

**Step 1: Understanding Auditing and Restrictions**

Auditing in a database involves tracking and logging activities to monitor changes to sensitive data, ensuring accountability and improving security. Restrictions, on the other hand, control access to data based on user roles, ensuring that only authorized users can perform certain operations.

**Step 2: Objectives of Auditing and Restrictions in Your Project**

1. **Logging Changes to Sensitive Data**: Track modifications to sensitive information like applicant details, status, and notification responses.
2. **Tracking User Actions**: Log actions performed by users, such as inserting, updating, or deleting records, for accountability.
3. **Enforcing Restrictions Based on Roles**: Restrict access to data and actions based on user roles to ensure that users can only access or modify data they're authorized to.

**Step 3: Auditing Changes to Sensitive Data**

To implement auditing, we can use triggers that log changes to sensitive data, such as applicant details, into an audit table. The audit table will store the following information:

* The action performed (INSERT, UPDATE, DELETE).
* The user who performed the action.
* The date and time of the action.
* The old and new values for the affected data.

CREATE TABLE PE\_Applicant\_Audit (

AuditID NUMBER PRIMARY KEY,

ApplicantID NUMBER,

FirstName VARCHAR2(100),

LastName VARCHAR2(100),

DateOfBirth DATE,

Address VARCHAR2(255),

PhoneNumber VARCHAR2(20),

Email VARCHAR2(100),

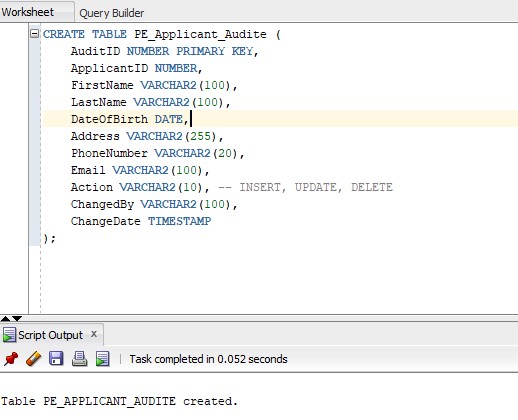
Action VARCHAR2(10), -- INSERT, UPDATE, DELETE

ChangedBy VARCHAR2(100),

ChangeDate TIMESTAMP

);

**Application:**

****

**Code: Trigger for Auditing Changes (PE\_Applicant\_Audit\_Trigger)**

CREATE OR REPLACE TRIGGER audit\_applicant\_changes

AFTER INSERT OR UPDATE OR DELETE ON PE\_Applicant

FOR EACH ROW

BEGIN

IF INSERTING THEN

INSERT INTO PE\_Applicant\_Audit (

AuditID, ApplicantID, FirstName, LastName, DateOfBirth, Address, PhoneNumber, Email,

Action, ChangedBy, ChangeDate

) VALUES (

PE\_Applicant\_Audit\_SEQ.NEXTVAL, :NEW.ApplicantID, :NEW.FirstName, :NEW.LastName, :NEW.DateOfBirth,

:NEW.Address, :NEW.PhoneNumber, :NEW.Email, 'INSERT', USER, SYSDATE

);

ELSIF UPDATING THEN

INSERT INTO PE\_Applicant\_Audit (

AuditID, ApplicantID, FirstName, LastName, DateOfBirth, Address, PhoneNumber, Email,

Action, ChangedBy, ChangeDate

) VALUES (

PE\_Applicant\_Audit\_SEQ.NEXTVAL, :NEW.ApplicantID, :NEW.FirstName, :NEW.LastName, :NEW.DateOfBirth,

:NEW.Address, :NEW.PhoneNumber, :NEW.Email, 'UPDATE', USER, SYSDATE

);

ELSIF DELETING THEN

INSERT INTO PE\_Applicant\_Audit (

AuditID, ApplicantID, FirstName, LastName, DateOfBirth, Address, PhoneNumber, Email,

Action, ChangedBy, ChangeDate

) VALUES (

PE\_Applicant\_Audit\_SEQ.NEXTVAL, :OLD.ApplicantID, :OLD.FirstName, :OLD.LastName, :OLD.DateOfBirth,

:OLD.Address, :OLD.PhoneNumber, :OLD.Email, 'DELETE', USER, SYSDATE

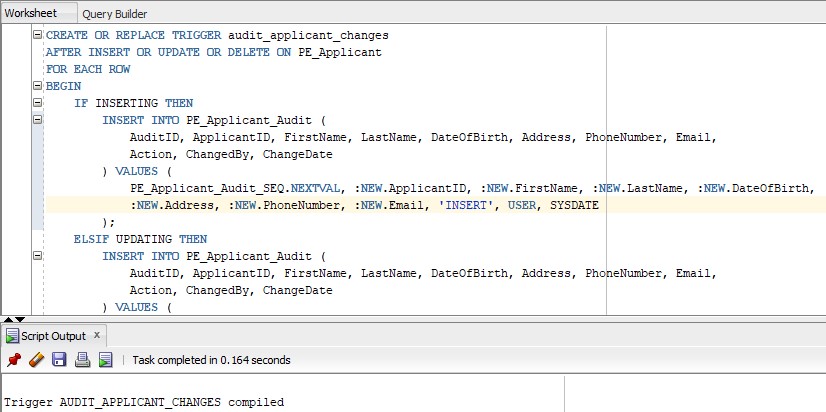
);

END IF;

END;

/

**Application:**

****

**Step 4: Tracking User Actions for Accountability**

**We can implement auditing mechanisms that log who performed a particular action. This can be achieved through:**

1. **The USER Function: The USER function in Oracle returns the username of the currently connected database user. We can use this to track which user performed a specific action.**
2. **Tracking via Logs: Record each operation performed by users on sensitive tables and associate them with their identity.**

**Code: Track User Actions in the Audit Table**

**The same audit\_applicant\_changes trigger will log the user who performed the insert, update, or delete action via the USER function.**

**Step 5: Implementing Restrictions Based on Roles**

**To restrict access to sensitive data based on user roles, we can define roles and assign privileges accordingly.**

**Code: Creating Roles and Assigning Privileges**

-- Create a role for admins

CREATE ROLE AdminRole;

-- Create a role for regular users

CREATE ROLE UserRole;

-- Assign privileges to AdminRole

GRANT SELECT, INSERT, UPDATE, DELETE ON PE\_Applicant TO AdminRole;

-- Assign privileges to UserRole (only select)

GRANT SELECT ON PE\_Applicant TO UserRole;

-- Assign roles to users

GRANT AdminRole TO admin\_user;

GRANT UserRole TO regular\_user;

**Step 6: Applying Triggers and Packages to Enforce Auditing and Restrictions**

**We have already implemented the auditing trigger (audit\_applicant\_changes). Restrictions are applied using the GRANT statements above, based on roles.**

**Step 7: Documenting the Benefits of Auditing and Restrictions**

1. **Improved Security: Auditing tracks any unauthorized or accidental changes to sensitive data, ensuring accountability and reducing the risk of unauthorized modifications.**
2. **Accountability: Every action is logged with the user's identity, helping track who made changes to sensitive data.**
3. **Role-Based Access Control: Access to data is restricted based on user roles, ensuring that users only perform actions they are authorized to.**
4. **Compliance: Auditing ensures that the system complies with organizational or regulatory requirements to track user actions.**

**Step 8: Testing Auditing and Restrictions**

* **Test 1: Perform an INSERT on the PE\_Applicant table and check if the change is logged in the PE\_Applicant\_Audit table.**
* **Test 2: Perform an UPDATE on the PE\_Applicant table and ensure that the audit table records the old and new values.**
* **Test 3: Perform a DELETE and verify that the action is recorded.**
* **Test 4: Test role restrictions by logging in as a user with the UserRole and trying to execute an INSERT or UPDATE.**

**Scope and Limitations of Implemented Features**

* Below is a detailed breakdown of the scope of each implemented feature (triggers, cursors, functions, packages, and auditing), along with any limitations or constraints introduced by these implementations.

**Scope and Limitations of Implemented Features**

Below is a detailed breakdown of the scope of each implemented feature (triggers, cursors, functions, packages, and auditing), along with any limitations or constraints introduced by these implementations.

**1. Triggers**

**Scope:**

* **Action Logging**: Triggers track INSERT, UPDATE, and DELETE operations on the PE\_Applicant table and log changes to the PE\_Applicant\_Audit table.
* **Enforcement of Business Logic**: Triggers can enforce business rules, such as automatic updates of statuses or changes to applicant data based on certain conditions.
* **Audit Functionality**: Triggers store detailed information about database changes, including user actions and the specific changes made, in an audit table for tracking and accountability.

**Limitations:**

* **Performance Overhead**: Triggers add processing overhead to each database transaction, potentially slowing down operations like inserts, updates, and deletes due to logging.
* **Complexity**: Complex triggers involving multiple conditions and actions can become difficult to maintain, especially as the database schema evolves.
* **Limited Scope**: Triggers can only operate on changes made within the database. Any changes outside of the database (such as through external APIs) won't be captured unless they are routed through the database.

**2. Cursors**

**Scope:**

* **Row-by-Row Processing**: Cursors are used to fetch data row by row from the PE\_Applicant table. This approach allows for detailed processing of each applicant (e.g., updating statuses or deleting inactive applicants).
* **Flexible Querying**: Cursors allow for more complex querying that cannot be easily handled by single SQL statements. This includes fetching a large set of data based on dynamic conditions.
* **Complex Operations**: Cursors are useful for performing operations like updating or deleting records based on logic that requires processing each record individually.

**Limitations:**

* **Memory Usage**: Using cursors with large result sets can consume significant memory, especially if the cursor fetches many rows at once.
* **Slower Execution**: Cursors can be slower compared to set-based operations in SQL. If possible, set-based updates should be preferred for performance reasons.
* **Not Ideal for Large Datasets**: If the dataset is large, using cursors for row-by-row operations can significantly degrade performance. In such cases, SQL operations should be optimized or considered for batch processing.

**3. Functions**

**Scope:**

* **Encapsulation of Logic**: Functions encapsulate specific tasks or calculations, such as deriving a certain value from multiple database fields or performing complex data manipulations.
* **Reusability**: Functions allow for the reuse of common logic across multiple database queries, ensuring that business logic is centralized and easy to maintain.
* **Modularization**: Functions help divide the logic into smaller, reusable units, improving readability and modularity in the codebase.

**Limitations:**

* **Performance Considerations**: While functions help with modularity, using them excessively or for complex calculations in queries can affect performance, especially if called repeatedly in large result sets.
* **Limited by SQL Constraints**: Functions cannot execute DML operations like INSERT, UPDATE, or DELETE unless they are specifically designed to do so. They also cannot modify the database state directly in most cases unless they are wrapped in a procedure.
* **Error Handling**: Functions that don't handle exceptions properly may cause issues during execution, especially if they are used in large-scale operations or embedded within other queries.

**4. Packages**

**Scope:**

* **Grouping Related Logic**: Packages allow for the grouping of related procedures, functions, and variables, providing better organization and modularity.
* **Data Encapsulation**: Packages allow the encapsulation of complex logic and state within a single unit, which is easier to manage and update.
* **Security and Access Control**: Packages enable the use of private and public procedures, giving control over which parts of the code are exposed to the outside world.

**Limitations:**

* **Complexity**: As packages grow in size, they can become difficult to maintain, especially if they contain many procedures and functions. It can also make debugging harder if not properly structured.
* **Dependency Management**: If there are interdependencies between different parts of a package, any changes in one procedure or function might require updating or testing the entire package.
* **Versioning**: Managing versions of packages can be difficult, especially when changes need to be rolled out across different environments.

**5. Auditing**

**Scope:**

* **Change Logging**: Auditing captures any changes to sensitive data (e.g., updates to applicant information or status changes), ensuring that every modification is recorded.
* **User Accountability**: By logging the user who made changes, auditing helps track who performed what actions on the database, providing accountability.
* **Monitoring Security and Compliance**: Auditing ensures that all activities on sensitive tables are monitored and tracked, improving database security and ensuring compliance with internal or regulatory requirements.

**Limitations:**

* **Storage Overhead**: Auditing logs can grow rapidly, especially in high-transaction environments. This requires additional storage and might necessitate log rotation or archiving strategies to manage the size.
* **Performance Impact**: Auditing every change in real-time can introduce performance overhead. Auditing triggers, in particular, can slow down the database transactions because they add extra logic to each modification.
* **Selective Auditing**: While auditing mechanisms are effective for tracking changes, they require careful configuration to ensure that the right data is captured. Auditing too much data can lead to unnecessary complexity and performance degradation.

**Conclusion: Summary of Scope and Limitations**

* **Scope**: Each implemented feature (triggers, cursors, functions, packages, auditing) aims to enhance functionality, improve data integrity, ensure accountability, and modularize the codebase.
* **Limitations**:
  + **Triggers**: Performance overhead and maintenance complexity.
  + **Cursors**: Memory usage and slower execution for large datasets.
  + **Functions**: Potential performance hits when overused, especially for complex logic.
  + **Packages**: Can become complex as they grow, with challenges around versioning and dependency management.
  + **Auditing**: Storage overhead and potential performance impact when logging high volumes of data.