Grundlagen von PL/SQL

Architektur von PL/SQL

- **Blockstruktur: *PL/SQL besteht aus Blöcken, die aus drei Teilen bestehen*:**
  - **Declarative Part**: *Hier werden Variablen, Konstanten und Cursor deklariert.*
  - **Executable Part**: *Hier wird der eigentliche Code ausgeführt.*
  - **Exception Handling Part**: *Optional, für Fehlerbehandlung.*

Variablen

- **Deklaration**: `VARIABLE_NAME datatype [:= value];`
  - Beispiele:

```
DECLARE
    v_number NUMBER := 10;
    v_name VARCHAR2(50) := 'John';
BEGIN
    -- Code hier
END;
```

Zuweisungen

- **Zuweisen von Werten**: `variable := value ;`
  - Beispiel:

```
v_number := 20;
```

Kontrollstrukturen

- `IF / THEN / ELSE` :

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

Schleifen:

- ## Simple `LOOP` :

```
LOOP
    -- Statements
    EXIT WHEN condition;
END LOOP;
```

- ## `FOR` Loop:

```
FOR i IN 1..10 LOOP
    -- Statements
END LOOP;
```

- ## `WHILE` Loop:

```
WHILE condition LOOP
    -- Statements
END LOOP;
```

Anonyme Blöcke

- **Definition**: Ein anonymer Block ist ein PL/SQL-Block, der nicht benannt ist und direkt ausgeführt wird.

```
BEGIN
    -- Statements hier
END;
```

Stored FUNCTIONS

- **Eigenschaften**: Funktionen, die in der Datenbank gespeichert und von SQL oder PL/SQL aufgerufen werden können.
- **Erstellen**:

```
CREATE FUNCTION function_name (param1 datatype, ...)
RETURN datatype IS
BEGIN
    -- Statements
    RETURN return_value;
END;
```

- **Aufruf**: `SELECT function_name(param1, ...) FROM dual;`

Stored PROCEDURES

- **Eigenschaften**: Ähnlich wie Funktionen, aber geben keinen Wert zurück, sondern führen Operationen durch.
- **Erstellen**:

```
CREATE PROCEDURE procedure_name (param1 datatype, ...)
IS
BEGIN
    -- Statements
END;
```

- **Aufruf**: `EXECUTE procedure_name(param1, ...);`

Datenbankzugriff

Lesen einer Zeile mittels `SELECT ... INTO`

- ### In skalare Variablen:

```
DECLARE
    v_emp_id NUMBER;
    v_emp_name VARCHAR2(50);
BEGIN
    SELECT employee_id, last_name INTO v_emp_id, v_emp_name
    FROM employees WHERE employee_id = 100;
END;
```

- ### Mit `%TYPE` und `%ROWTYPE`:

  - `%TYPE` : Behält den **Datentyp der Spalte** bei:

    ```
    v_emp_id employees.employee_id%TYPE;
    ```

  - `%ROWTYPE` : Für eine ganze **Zeile als Record**:

    ```
    v_emp_row employees%ROWTYPE;
    SELECT * INTO v_emp_row FROM employees WHERE employee_id = 100;
    ```

Iterieren über Ergebnislisten mittels Cursor

Manuelles OPEN / FETCH / CLOSE

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, last_name FROM employees;
    v_emp_id NUMBER;
    v_emp_name VARCHAR2(50);
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_emp_id, v_emp_name;
        EXIT WHEN emp_cursor%NOTFOUND;
        -- Code hier
    END LOOP;
    CLOSE emp_cursor;
END;
```

CURSOR-FOR Schleife

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, last_name FROM employees;
BEGIN
    FOR emp_rec IN emp_cursor LOOP
        -- Code hier mit emp_rec.employee_id und emp_rec.last_name
    END LOOP;
END;
```

Trigger

DML Trigger

- **Statement-Level Trigger**: Führt ein Mal pro SQL-Anweisung aus.
- **Row-Level Trigger**: Führt für jede betroffene Zeile aus.
- *Erstellen eines Triggers*:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE/AFTER INSERT/UPDATE/DELETE ON table_name
FOR EACH ROW
BEGIN
    IF :NEW.column_name IS NOT NULL THEN
        -- Logik hier
    END IF;
END;
```

  - `:OLD` und `:NEW` sind spezielle Pseudorecords, die auf alte und neue Werte zugreifen.