

## Iterative processing with loops

### Nested IF statement:

You can nest an IF statement within another IF statement as shown below:

```
<<label>> LOOP  
    statements;  
END LOOP loop_label;
```



## Iterative processing with loops

```
DECLARE
  I_counter NUMBER := 0;
BEGIN
  LOOP
    I_counter := I_counter + 1;
    IF I_counter > 3 THEN
      EXIT;
    END IF;
    dbms_output.put_line( 'Inside loop: ' || I_counter ) ;
  END LOOP;
  -- control resumes here after EXIT
  dbms_output.put_line( 'After loop: ' || I_counter );
END;
```



## FOR LOOP

PL/SQL FOR **LOOP** executes a sequence of statements a specified number of times. The PL/SQL FOR LOOP statement has the following structure:

```
FOR index IN lower_bound .. upper_bound  
LOOP  
    statements;  
END LOOP;
```



## FOR LOOP

```
BEGIN  
  FOR I_counter IN 1..5  
  LOOP  
    DBMS_OUTPUT.PUT_LINE( I_counter );  
  END LOOP;  
END;
```



## FOR LOOP

```
DECLARE  
  I_step PLS_INTEGER := 2;  
BEGIN  
  FOR I_counter IN 1..5 LOOP  
    dbms_output.put_line (I_counter*I_step);  
  END LOOP;  
END;
```



## FOR LOOP

```
DECLARE  
  I_counter PLS_INTEGER := 10;  
BEGIN  
  FOR I_counter IN 1.. 5 loop  
    DBMS_OUTPUT.PUT_LINE (I_counter);  
  end loop;  
  -- after the loop  
  DBMS_OUTPUT.PUT_LINE (I_counter);  
END;
```



## WHILE loop

Here is the syntax for the **WHILE loop** statement:

```
WHILE condition  
LOOP  
    statements;  
END LOOP;
```



## WHILE loop

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





## FOR LOOP

### WHILE loop example terminated by EXIT WHEN statement:

The following example is the same as the one above except that it has an additional EXITWHEN statement.



## WHILE loop

```
DECLARE
  n_counter NUMBER := 1;
BEGIN
  WHILE n_counter <= 5
  LOOP
    DBMS_OUTPUT.PUT_LINE( 'Counter : ' || n_counter );
    n_counter := n_counter + 1;
    EXIT WHEN n_counter = 3;
  END LOOP;
END;
```



## CONTINUE statement

The **CONTINUE** statement allows you to exit the current loop iteration and immediately continue on to the next iteration of that loop.

**The CONTINUE statement has a simple syntax:**

```
CONTINUE;
```



## CONTINUE statement

Typically, the **CONTINUE** statement is used within an IF THEN statement to exit the current loop iteration based on a specified condition as shown below:

```
IF condition THEN  
    CONTINUE;  
END IF;
```



## CONTINUE statement

```
BEGIN
FOR n_index IN 1 .. 10
LOOP
  -- skip odd numbers
  IF MOD( n_index, 2 ) = 1 THEN
    CONTINUE;
  END IF;
  DBMS_OUTPUT.PUT_LINE( n_index );
END LOOP;
END;
```



## CONTINUE statement

### CONTINUE WHEN statement:

The **CONTINUE WHEN** statement exits the current loop iteration based on a condition and immediately continue to the next iteration of that loop.

The syntax of **CONTINUE WHEN** statement is:

**CONTINUE WHEN** condition;



## CONTINUE statement

```
BEGIN
FOR n_index IN 1 .. 10
LOOP
  -- skip even numbers
  CONTINUE
WHEN MOD( n_index, 2 ) = 0;
  DBMS_OUTPUT.PUT_LINE( n_index );
END LOOP;
END;
```



## Exception

PL/SQL treats all errors that occur in an anonymous block, procedure, or function as **exceptions**.

The exceptions can have different causes such as coding mistakes, bugs, even hardware failures.

It is not possible to anticipate all potential exceptions, however, you can write code to handle exceptions to enable the program to continue running as normal.





# Exception

The code that you write to handle exceptions is called an **exception handler**.

```
BEGIN
  -- executable section
  ...
  -- exception-handling section
EXCEPTION
  WHEN e1 THEN
    -- exception_handler1
  WHEN e2 THEN
    -- exception_handler1
  WHEN OTHERS THEN
    -- other_exception_handler
END;
```



## Exception

```
DECLARE
c_id student.id%type := 2;
c_name student.Name%type;

BEGIN
SELECT name, id INTO c_name, c_id
FROM student
WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
DBMS_OUTPUT.PUT_LINE ('id: ' || c_id);
EXCEPTION
WHEN no_data_found THEN
dbms_output.put_line('No such student!');
WHEN others THEN
dbms_output.put_line('Error!');
END;
```



## Records

A PL/SQL **record** is a composite data structure which consists of multiple fields; each has its own value. The following picture shows an example record that includes first name, last name, email, and phone number:



## Records

PL/SQL **record** helps you simplify your code by shifting from field-level to record-level operations.

PL/SQL has three types of records: **table-based**, **cursor-based**, programmer-defined.

Before using a record, you must declare it.

```
DECLARE  
    record_name table_name%ROWTYPE;
```



## Records

```
CREATE TABLE persons (  
    person_id NUMBER GENERATED BY DEFAULT  
    AS IDENTITY,  
    first_name VARCHAR2( 50 ) NOT NULL,  
    last_name VARCHAR2( 50 ) NOT NULL,  
    primary key (person_id)  
);
```



## Records

```
DECLARE
  r_person persons%ROWTYPE;

BEGIN
  -- assign values to person record
  r_person.person_id := 1;
  r_person.first_name := 'John';
  r_person.last_name := 'Doe';

  -- insert a new person
  INSERT INTO persons VALUES r_person;
END;
```



## Records

```
DECLARE
  r_person persons%ROWTYPE;

BEGIN
  -- get person data of person id 1
  SELECT * INTO r_person
  FROM persons
  WHERE person_id = 1;

  -- change the person's last name
  r_person.last_name := 'Smith';

  -- update the person
  UPDATE persons
  SET ROW = r_person
  WHERE person_id = r_person.person_id;
END;
```

