PL/SQL (II)

Ș.L. Dr. Ing. Ciprian-Octavian Truică

ciprian.truica@upb.ro

As. Drd. Ing. Alexandru Petrescu

Alex.petrescu@upb.ro

Overview

Variables and constants

Data types

Operators

Control statements

Variables

```
v_valoare NUMBER(15) NOT NULL := 0;
v_data_achizitie DATE DEFAULT SYSDATE;
v_material VARCHAR2(15) := 'Matase';
c_valoare CONSTANT NUMBER := 100000;
v_stare VARCHAR2(20) DEFAULT 'Buna';
v_clasificare BOOLEAN DEFAULT FALSE;
int_an_luna INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO
MONTH;
```

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

- When declaring a **variable**, **storage space** is allocated for a value of a specified **data type**
- The **storage location** is **marked** with a name so it can be **referenced**
- Objects must be declared **before** referencing them
- Declarations can appear in the **declarative** part of any Block / Subprogram / Package

Datatypes:

- **Scalar**: Contain a **single** value (No internal components)
- **Composit**: **Collections** / **Records**

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

- **variable_name** - Name of the variable that you are declaring
- **CONSTANT –** Imposes that the value cannot be changed after assignment
- **datatype** - Name of a scalar data type, including any qualifiers for size, precision, and character or byte semantics
- **NOT NULL** - Imposes the NOT NULL constraint on the variable
- **:= / DEFAULT –** Assigning the initial value
- **initial_value** – Immediate value or expression

Variables

```
v_valoare NUMBER(15) NOT NULL := 0;
v_data_achizitie DATE DEFAULT SYSDATE;
v_material VARCHAR2(15) := 'Matase';
c_valoare CONSTANT NUMBER := 100000;
v_stare VARCHAR2(20) DEFAULT 'Buna';
v_clasificare BOOLEAN DEFAULT FALSE;
int_an_luna INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO
MONTH;
```

NOT NULL Constraint

An item can acquire this **constraint** either **Implicitly** (from its data type) or **Explicitly**

The **default initial value** for a scalar variable is NULL.

A scalar variable declaration that specifies **NOT NULL** (either implicitly or explicitly) **must** assign **an initial value** to the variable.

PL/SQL treats any **zero-length string** as a **NULL** value, this includes:

- Values returned by character functions
- BOOLEAN expressions.

Initial Values of Variables and Constants

If the declaration is

- **A variable**: the initial value is optional unless the NOT NULL constraint is specified
- **A constant**: the initial value is required

If the declaration is in a **block** or **subprogram**:

- The initial value is assigned to the variable or constant **every time** control passes to the block or subprogram

If the declaration is in a **package specification**

- The initial value is assigned to the variable or constant for **each session** (whether the variable or constant is public or private)

Initial Values of Variables and Constants

To specify the **initial value** use

- The assignment operator (**:=**) followed by an expression
- The keyword **DEFAULT** followed by an expression

The expression can include **previously declared**

- **Constants**
- **Initialized variables**
- **Functions results**

If **no initial value** is specified for a variable, then a value **must be assigned before using it** in any other context.

Example

```
DECLARE
    no_months constant BINARY_INTEGER:= 12;
    emp_name varchar2(45);
    emp_salary employees.salary%TYPE;
    anual_salary float(10);
BEGIN
    SELECT first_name || ' ' || last_name, salary
    INTO emp_name, emp_salary
    FROM employees WHERE employee_id = &id;

    anual_salary := emp_salary * no_months;
    dbms_output.put_line(rpad(emp_name, 20) || '-' || lpad(anual_salary, 10));
END;
/
```

Data Types

Initial Values of Variables and Constants

Every PL/SQL **constant**, **variable**, **parameter**, and **function** (block that returns a value) has a data type.

The data type determines

- The storage format
- Valid values
- Operations

The data type can be assiged

- **Explicit** – by naming the data type used (e.g. **NUMBER**, **VARCHAR2**, etc.)
- **Implicit** – by using attributes (**%TYPE**, **%ROWTYPE**)


Find YOUR Type
(using %TYPE)

Declaring **Variables** using **%TYPE**

The **%TYPE** attribute permits the declaration of a data item of the same data type as a previously declared **variable** or **column** (**without** knowing what that type is).

If the declaration of the **referenced** item changes, then the declaration of the **referencing** item changes accordingly.

The syntax of the declaration is: referencing_item referenced_item**%TYPE**;

Declaring **Variables** using **%TYPE**

The referencing item **inherits** the following from the **referenced** item:

- **Data type** and **size**
- **Constraints** (unless the referenced item is a column)

The referencing item **does not inherit** the initial **value** of the referenced item: If the referencing item specifies or inherits the **NOT NULL** constraint, an initial **value** for it **must be specified**.

The **%TYPE** attribute is particularly useful when declaring variables to hold **database values.**

Example

```
DECLARE
    a INT NOT NULL := 4;
    b a%TYPE := 3;

    c employees.employee_id%TYPE := 4;
    d c%TYPE default 3;
BEGIN
    dbms_output.put_line(a);
    dbms_output.put_line(b);

    dbms_output.put_line(c);
    dbms_output.put_line(d);
END;
/
```

Scalar data types

- Scalar data types store values with **no internal components**
- A scalar data type can have **subtypes**
- A **subtype** is a data type that is a **subset** of another data type (its base type)
- A **subtype** has the same valid **operations** as its base type
- A data type and its subtypes comprise a **data type family**
- PL/SQL **predefines** many types and subtypes
- The **predefined** data types are available in the **STANDARD package**
- Developers can also define **their own subtypes** (**UDST** – User Defined Subtypes)

Scalar data types

The PL/SQL **scalar** data types are:

- The SQL data types
- BOOLEAN
- PLS_INTEGER
- BINARY_INTEGER
- REF CURSOR
- User-defined subtypes

SQL data types

- **Character** Data Types
- **Number** Data Types
- **Datetime** and **Interval** Data Types
- **RAW** and **LONG RAW** Data Types
- Large Object (**LOB**) Data Types
- **Rowid** Data Types


E

Character Data Types

CHAR : specifies a **fixed**-length character string in the database character set

VARCHAR2 : specifies a **variable**-length character string in the database character set.

ASCII VS **Unicode**

NCHAR : specifies a **fixed**-length character string in the national character set

NVARCHAR2 : specifies a **variable**-length character string in the national character set.


6
24
4
4
6
6
4
24

Number Data Types

The NUMBER data type stores zero as well as positive and negative fixed **numbers**.

NUMBER(p, s)

- p is the **precision**: the **maximum** number of **significant decimal digits**
- s is the **scale**: the number of digits from the **decimal point** to the least significant digit.

Number Data Types

The **FLOAT** data type is a **subtype** of **NUMBER**. It can be specified with or without **precision**. Scale cannot be specified, is interpreted from the data.

FLOAT(p) - p is the precision: the maximum number of **significant decimal digits**

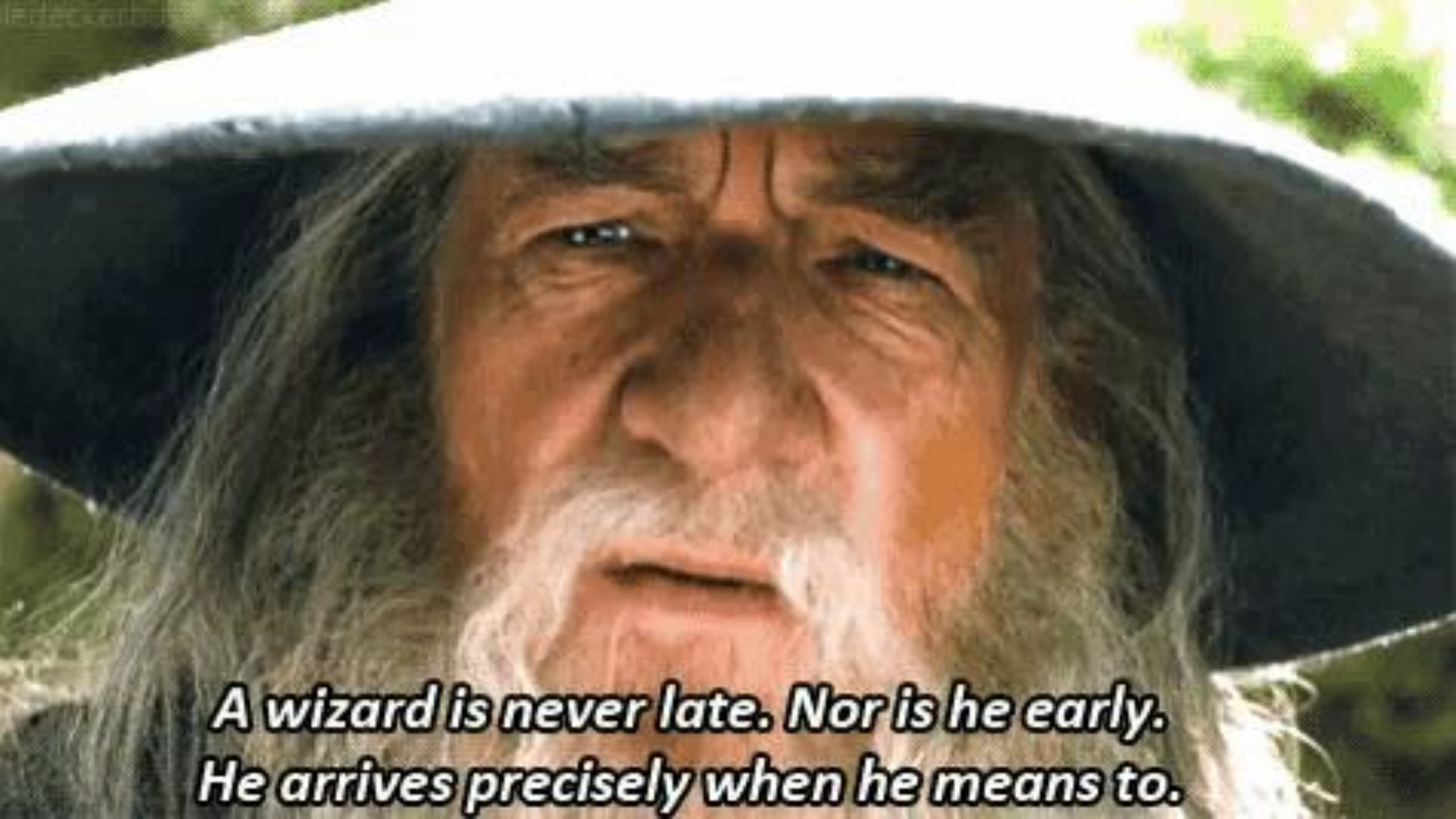
Floating-point numbers can have a decimal point anywhere from the first to the last digit or can have no decimal point at all.

Oracle Database provides two numeric data types exclusively for floating-point numbers:

- **BINARY_FLOAT** is a **32-bit**, single-precision floating-point number data type. Each **BINARY_FLOAT** value requires **4 bytes**.
- **BINARY_DOUBLE** is a **64-bit**, double-precision floating-point number data type. Each **BINARY_DOUBLE** value requires **8 bytes**.

FLOAT vs **BINARY_FLOAT** vs **BINARY_DOUBLE**

- **BINARY_FLOAT** & **BINARY_DOUBLE** data types take advantage of **hardware acceleration**, therefore, they have better performance for numerical computations.
- **BINARY_FLOAT** & **BINARY_DOUBLE** data types can store smaller / larger numbers than **FLOAT** type.
- **BINARY_FLOAT** & **BINARY_DOUBLE** data types store only **approximate** values, while the **FLOAT** data type stores **exact** values.
- **BINARY_FLOAT** & **BINARY_DOUBLE** data types are suitable for the **scientific calculations** but **not** suitable for **financial calculations**.


A wizard is never late. Nor is he early.
He arrives precisely when he means to.

Datetime and **Interval** Data Types

The **DATE** data type stores **date** and **time** information

- Oracle stores the following information for a **DATE**: year, month, day, hour, minute, and second

The **TIMESTAMP** data type is an **extension** of the **DATE** data type. It stores all information from of the **DATE** data type plus **fraction of seconds**. This data type is useful for

- Storing **precise** time values
- **Collecting** and **Evaluating** date information across **geographic** regions

timestamp_var **TIMESTAMP** [(fractional_seconds_precision)]

timestamp_with_local_var **TIMESTAMP** [(fractional_seconds_precision)] **WITH LOCAL TIME ZONE**

- **TIMESTAMP WITH LOCAL TIME ZONE** is another variant of **TIMESTAMP** that is sensitive to time zone information.
- **fractional_seconds_precision** optionally specifies the **number of digits** Oracle stores in the fractional part of the SECOND datetime field.

Datetime and **Interval** Data Types

INTERVAL YEAR TO MONTH stores a period of time using the **YEAR** and **MONTH** datetime fields. This data type is useful for representing the difference between two datetime values when only the year and month values are significant.

INTERVAL YEAR [(year_precision)] **TO MONTH**

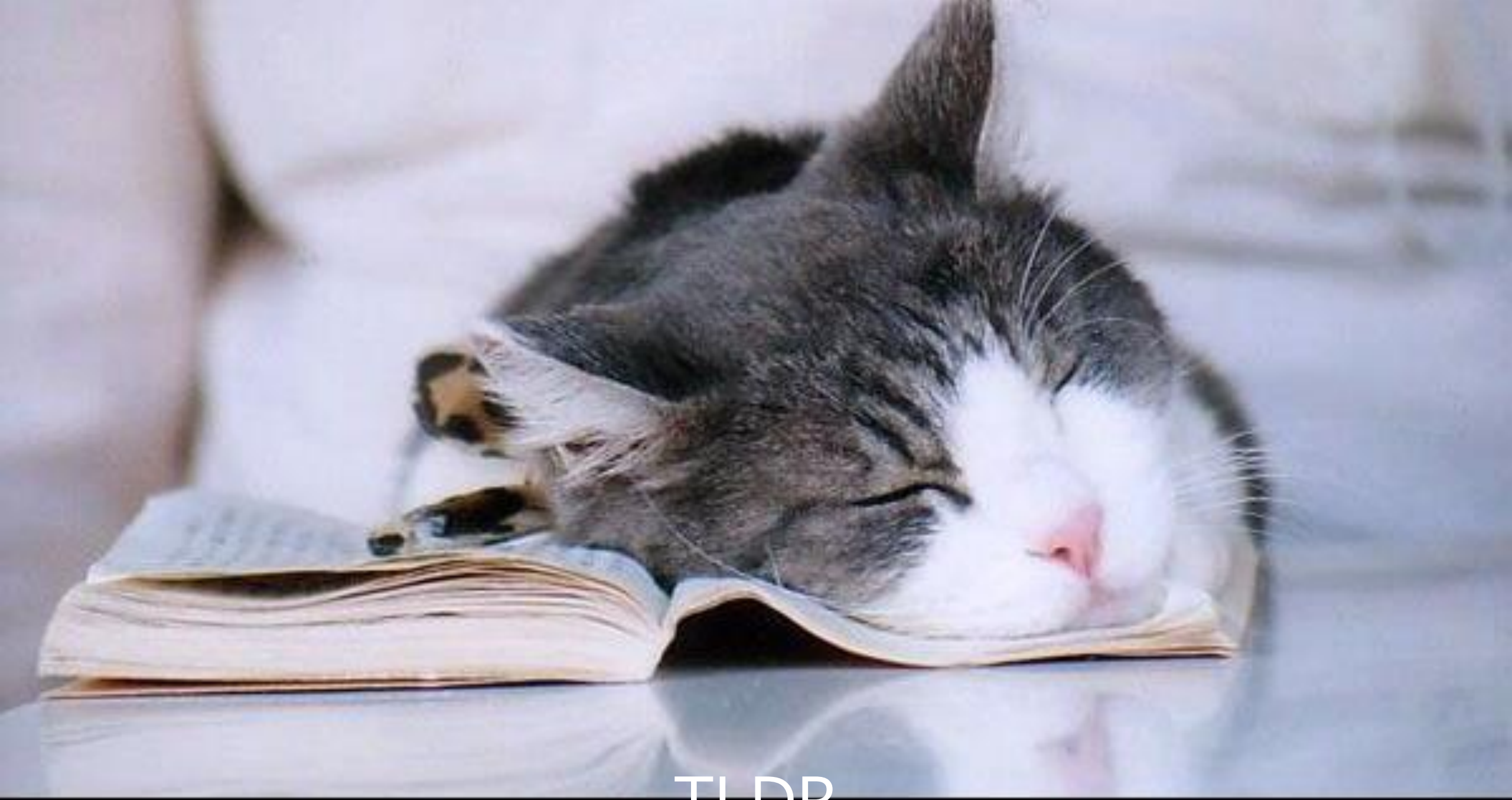
- **year_precision** is the number of **digits** in the **YEAR** datetime field

Datetime and **Interval** Data Types

INTERVAL DAY TO SECOND stores a period of time in terms of days and seconds. This data type is useful for representing the precise difference between two datetime values

INTERVAL DAY [(day_precision)] **TO SECOND** [(fractional_seconds_precision)]

- **day_precision** is the number of digits in the **DAY** datetime field. Accepted values are 0 to 9. The default is **2**.
- **fractional_seconds_precision** is the number of digits in the fractional part of the **SECOND** datetime field. Accepted values are 0 to 9. The default is **6**.


TLDR

RAW and **LONG RAW** Data Types

The **RAW** and **LONG RAW** data types store data that is **not to be explicitly converted** by Oracle Database when moving data between different systems. These data types are intended for **binary data** or **byte strings**. Oracle **strongly recommends** that you convert **LONG RAW** columns to binary **LOB** (BLOB) columns. **LOB** columns are subject to far **fewer restrictions** than **RAW** columns. The built-in LOB data types:

- BLOB
- CLOB
- NCLOB (stored internally)
- BFILE (stored externally)

LOBs can store **large** and **unstructured** data such as text, image, video, and spatial data.

BLOB vs **CLOB** vs **NCLOB** Data Types

The **BLOB** data type stores **unstructured binary large objects**. The **CLOB** data type stores **single-byte** (ASCII) character data or **multibyte** (Unicode) character data. The **NCLOB** data type stores **Unicode** data.

BLOB objects can be thought of as bitstreams with no character set semantics.

CLOB and **NCLOB** support:

- **Fixed-width** character sets (uses the database character set)
- **Variable-width** character sets (uses the database character set)

ALL of Them store binary data up to (4 GB - 1B) * LOB_CHUNK_SIZE

- **LOB_CHUNK_SIZE** is **equal** to the database **block size**

BFILE Data Types

The **BFILE** data type enables access to binary file **LOBs** that are stored in file systems **outside** Oracle Database. A **BFILE column** or **attribute** stores a **BFILE locator**.

BFILE locator serves as a **pointer** to a **binary file** on the **server file system**. The **locator** maintains the **directory name** and the **filename**.

Binary file **LOBs** do not participate in transactions and are **not recoverable**. The **operating system** provides file **integrity** and **durability**.

BFILE data can be up to 2^64-1 bytes (the operating system may impose restrictions on this maximum). The **database administrator** must **ensure** that the **external** file exists. **Oracle processes** have operating system **read** permissions on the file.

The **BFILE** data type enables **read-only** support of **large binary files**, such a file cannot be modified or replicated.


int *
int
int **
@programmer.meme

ROWID Data Types

Each row in the database has an **address**. The rows in **heap-organized tables** that are native to Oracle Database have **row addresses** called **rowids**.

A **rowid** row address can be examined by querying the **pseudocolumn ROWID**. Values of this **pseudocolumn** are **strings** representing the **address of each row**. These strings have the data type **ROWID**. **Rowids** contain the following information:

- The data block of the data file containing the row.
- The row in the data block.
- The database file containing the row
- The data object number which is an identification number assigned to every database segment.


PERHAPS

BOOLEAN Data Type

The PL/SQL data type **BOOLEAN** stores logical values: **TRUE** / **FALSE** / **NULL** (representing an unknown value)

Because SQL has **no** data type **equivalent** to **BOOLEAN**, it is **not** possible to:

- Assign a **BOOLEAN** value to a database **table column**
- **Select** or **fetch** the value of a database table column **into a BOOLEAN** variable
- Use a **BOOLEAN** value in a **SQL function**
- Use a **BOOLEAN** expression in a **SQL statement**, except
  - As an **argument** to a PL/SQL **function** invoked in a SQL query
  - In a PL/SQL **anonymous block**


Unlimited power!

PLS_INTEGER vs **BINARY_INTEGER** Data Types

The PL/SQL data types **PLS_INTEGER** and **BINARY_INTEGER** are identical. Both data types:

- Store **signed** integers in the range -2,147,483,648 through 2,147,483,647
- **Integers** are represented in **32 bits**

They have the following advantages over the **NUMBER** data type and **NUMBER subtypes**:

- Values require **less storage**
- Operations use **hardware arithmetic**
- Are faster than NUMBER operations which use **library arithmetic**


I MADE THIS

User-Defined PL/SQL Subtypes

Developers can define their **own subtypes** for restrincting: **Size** / **Precision** / **Scale**

```
SUBTYPE subtype_name IS base_type
  { precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

Basically **syntactic sugar** for some types. (see **INT** vs **NUMBER**)

Operators

Supported Operators in PL/SQL

- Arithmetic operators
- Logic operators
- Comparison operators
- SQL operators

Operator	Operation	Example
**	***Exponential***	val2 ** val1
+	***Identity***	+ val
-	***Negation***	- val
+	***Addition***	val1 + val2
-	***Subtraction***	val1 - val2
*	***Multiplication***	val1 * val2
/	***Division***	val1 / val2
||	***Concatenation***	val1 || val2

Arithmetic operators

Logical operators

Operator	Operation	Example
NOT	***Negation***	NOT val
AND	***Logical AND***	val1 AND val2
OR	***Logical OR***	val1 OR val2

OPERATOR	OPERATION	EXAMPLE
=	***Equal***	val2 = val1
<	***Less then***	val2 < val1
>	***Greater than***	val2 > val1
<=	***Less or equal than***	val1 <= val2
>=	***Greater or equal than***	val1 >= val2
<>	***Different than***	val1 <> val2
!=	***Different than***	val1 != val2
~=	***Different than***	val1 ~= val2
^=	***Different than***	val1 ^= val2

Comparison operators

OPERATOR	OPERATION	EXAMPLE
BETWEEN	***Verifies if a value is in a range***	val BETWEEN val1 AND val2
IN	***Verifies if a value is in a list***	val IN(val1, val2, …, val3)
LIKE	***Verifies a specified patter n***	val LIKE pattern
IS NULL	***Verifies if a value is NULL***	val IS NULL

SQL operators

PRECEDENCE	OPERATORS	OPERATION
1	**	Exponential
2	+, -	Identity, negation
3	*, /	Multiplication, division
4	+, -, ||	Addition, subtraction, concatenation
5	=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.	Comparison and SQL operators
6	NOT	Logical negation
7	AND	Conjunction
8	OR	Inclusion

Operators precedence

Symbol	Meaning
:=	Assignment operator
=>	Association operator
%	Attribute indicator
'	String delimiter
.	Component selector
( )	Expression or list delimiter
:	Host variable indicator
,	Item separator
<< >>	Label delimiter
/* */	Multiline comment delimiter
"	Quoted identifier delimiter
..	Range operator
@	Remote access indicator
--	Single-line comment indicator
;	Statement terminator

Other symbols

Control statements

Control Statements

- Conditional selection statements
- Loop statements
- Sequential control statements

Conditional selection statements

IF Statement

The condition can contain:

- Comparison operators
- Logical operators
- SQL operators

```
-- IF THEN Statement
IF condition THEN statements
END IF;

-- IF THEN ELSE Statement
IF condition THEN statements
ELSE else_statements
END IF;

-- IF THEN ELSIF Statement
IF condition_1 THEN statements_1
ELSIF condition_2 THEN statements_2
[ ELSIF condition_3 THEN statements_3 ]
...
[ ELSE else_statements ]
END IF;

-- Logically equivalent nested
-- IF THEN ELSE statements
IF condition_1 THEN statements_1;
ELSE
  IF condition_2 THEN statements_2;
  ELSE
    IF condition_3 THEN statements_3;
    ...
    END IF;
  END IF;
END IF;
```

```
DECLARE
    emp_name varchar2(45);
    emp_salary employees.salary%TYPE;
    emp_iddept employees.department_id%TYPE;
    avg_sal number(6,2);
    avg_sal_dept float(8);
BEGIN
    SELECT first_name ||' '|| last_name, salary, department_id
    INTO emp_name, emp_salary, emp_iddept
    FROM employees WHERE employee_id = &id;

    SELECT AVG(salary) INTO avg_sal FROM employees;

    SELECT AVG(salary) INTO avg_sal_dept FROM employees
    WHERE department_id= emp_iddept;

    IF emp_salary > avg_sal THEN
        dbms_output.put_line(emp_name || ' are sal > decat sal mediu din firma');
    END IF;

    IF emp_salary > avg_sal_dept THEN
        dbms_output.put_line(emp_name || ' are sal > sal mediu din departamentul in care lucreaza');
    ELSE
        dbms_output.put_line(emp_name || ' are sal <= sal mediu din departamentul in care lucreaza');
    END IF;
END;
/
```

```
DECLARE
    year_of_experience INTERVAL YEAR TO MONTH;
    expert INTERVAL YEAR TO MONTH;
    emp_name varchar2(45);
    emp_hiredate employees.hire_date%TYPE;
BEGIN
    expert := INTERVAL '15-1' YEAR TO MONTH;

    dbms_output.put_line(expert);
    dbms_output.put_line(sysdate());
    dbms_output.put_line(sysdate - expert);
    SELECT first_name ||' '|| last_name, hire_date
    INTO emp_name, emp_hiredate
    FROM employees WHERE employee_id = &id;

    IF emp_hiredate < sysdate - expert THEN
        dbms_output.put_line(emp_name ||' e expert');
    ELSE
        dbms_output.put_line(emp_name || ' nu e expert');
    END IF;
END;
/
```

Conditional selection statements

CASE Statement

Simple CASE

- The selector is an expression (typically a single variable)
- Each selector_value can be either a literal or an expression (cannot use NULL)
- The simple CASE statement runs the first statements for which selector_value equals selector
- Remaining conditions are not evaluated
- If no selector_value equals selector the CASE statement runs else_statements if they exist
- Raises the predefined exception CASE_NOT_FOUND otherwise

Searched CASE

- The searched CASE statement runs the first statements for which condition is true
- Remaining conditions are not evaluated

```
-- Simple CASE
CASE selector
WHEN selector_value_1 THEN statements_1
WHEN selector_value_2 THEN statements_2
...
WHEN selector_value_n THEN statements_n
[ ELSE else_statements ]
END CASE;

-- Searched CASE
CASE
WHEN condition_1 THEN statements_1
WHEN condition_2 THEN statements_2
...
WHEN condition_n THEN statements_n
[ ELSE else_statements ]
END CASE;
```

```
DECLARE
    emp_name varchar2(45);
    emp_iddept employees.department_id%TYPE;
    emp_reg regions.region_name%TYPE;
    emp_city locations.city%TYPE;
BEGIN
    SELECT first_name || ' ' || last_name, department_id
    INTO emp_name, emp_iddept
    FROM employees WHERE employee_id = &id;

    SELECT r.region_name, l.city INTO emp_reg, emp_city FROM regions r
        INNER JOIN countries c ON c.region_id = r.region_id
        INNER JOIN locations l ON l.country_id = c.country_id
        INNER JOIN departments d ON d.location_id = l.location_id
    WHERE d.department_id = emp_iddept;

    CASE emp_reg
        WHEN 'Europe' THEN
            IF emp_city = 'London' THEN
                dbms_output.put_line(emp_name || ' works in European HQ');
            ELSE
                dbms_output.put_line(emp_name || ' doesn''t works in HQ');
            END IF;
        WHEN 'Americas' THEN
            IF emp_city = 'Seattle' THEN
                dbms_output.put_line(emp_name || ' works in American HQ');
            ELSE
                dbms_output.put_line(emp_name || ' doesn''t works in HQ');
            END IF;
        ELSE
            dbms_output.put_line(emp_name || ' works outside Europe and America');
    END CASE;
END;
/
```

```
DECLARE
    emp_name varchar2(45);
    comm employees.commission_pct%TYPE;
BEGIN
    SELECT first_name ||' '|| last_name, commission_pct
    INTO emp_name, comm
    FROM employees WHERE employee_id = &id;

    CASE
        WHEN comm IS NULL THEN
            dbms_output.put_line(emp_name || ' nu are commision');
        WHEN comm >= 0 and comm < 0.2 THEN
            dbms_output.put_line(emp_name || ' are commision mic');
        WHEN comm >= 0.2 and comm < 0.3 THEN
            dbms_output.put_line(emp_name || ' are commision mediu');
        ELSE
            dbms_output.put_line(emp_name || ' are commision mare');
    END CASE;
END;
/
```

```
DECLARE
    emp_name varchar2(45);
    comm employees.commission_pct%TYPE;
BEGIN
    SELECT first_name || ' ' || last_name, commission_pct
    INTO emp_name, comm
    FROM employees WHERE employee_id = &id;

    IF comm IS NULL THEN
        dbms_output.put_line(emp_name || ' nu are commision');
    ELSIF comm >= 0 and comm < 0.2 THEN
        dbms_output.put_line(emp_name || ' are commision mic');
    ELSIF comm >= 0.2 and comm < 0.3 THEN
        dbms_output.put_line(emp_name || ' are commision mediu');
    ELSE
        dbms_output.put_line(emp_name || ' are commision mare');
    END IF;
END;
/
```

Loop Statements

Basic LOOP Statement

With **each iteration** of the loop:

- The statements are run
- Control returns to the top of the loop

To prevent an **infinite loop** and exit the loop:

- Use a statement
- Raised exception

```
-- Basic LOOP
[label] LOOP
  statements
  [ EXIT [ WHEN condition ] ]
  [ CONTINUE [ WHEN condition ] ]
END LOOP [label];
```

EXIT statements

The **EXIT** statement **ends** the **current iteration** of a loop **unconditionally**. It transfers control to the **end** of either the **current loop** or an **enclosing labeled** loop.

The **EXIT WHEN** statement ends the current iteration of a loop when the condition in its **WHEN** clause is **true**. The condition in the **WHEN** clause is evaluated **each time** control reaches the **EXIT WHEN** statement. The **EXIT WHEN** statement does nothing if the condition is not true.

CONTINUE statement

The **CONTINUE** statement **continues** the **current iteration** of a loop **unconditionally**. It transfers control to the **next iteration** of either the **current loop** or an **enclosing labeled** loop.

The **CONTINUE WHEN** statement continues the current iteration of a loop when the condition in its **WHEN** clause is true. The condition in the **WHEN** clause is evaluated each time control reaches the **CONTINUE WHEN** statement. The **CONTINUE WHEN** statement does nothing if the condition is not true.

```
DECLARE
    idx number NOT NULL := 1;
    deptname departments.department_name%TYPE;
    avg_sal_dept number(7,2);
BEGIN
    LOOP
        SELECT d.department_name, avg(e.salary)
        INTO deptname, avg_sal_dept
        FROM departments d
            INNER JOIN employees e
                ON d.department_id = e.department_id
        WHERE d.department_id = idx * 10
        GROUP BY d.department_name;

        dbms_output.put_line(deptname || ' ' || avg_sal_dept);
        idx := idx + 1;

        CONTINUE WHEN idx < 11;

        -- EXIT WHEN idx = 12;

    END LOOP;
END;
/
```

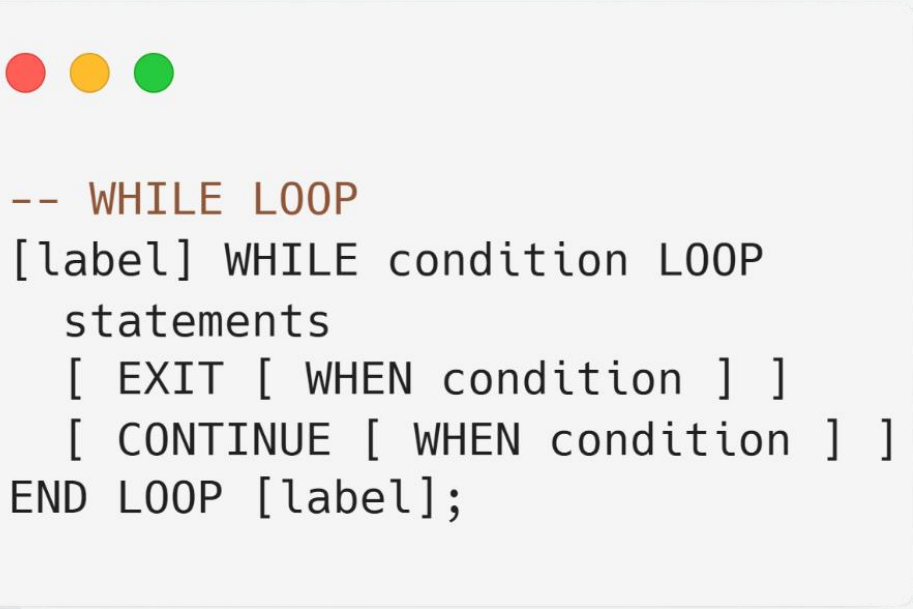
Loop Statements
WHILE LOOP Statement

The **WHILE** LOOP statement runs one or more statements **while** a condition is true.

If the condition is true the statements are run, then **control returns** to the **top of the loop** where condition is **evaluated** again.

The **WHILE** LOOP **stops** when the condition is false. A statement inside the loop must make the condition false or null to prevent an **infinite loop**.

An **EXIT**, **EXIT WHEN**, **CONTINUE**, or **CONTINUE WHEN** in the statements can cause the loop or the current iteration of the loop to end **early**.



```
-- WHILE LOOP
[label] WHILE condition LOOP
  statements
  [ EXIT [ WHEN condition ] ]
  [ CONTINUE [ WHEN condition ] ]
END LOOP [label];
```

```
DECLARE
    idx number := 1;
    deptname departments.department_name%TYPE;
    avg_sal_dept number(7,2);
BEGIN
    WHILE idx < 12
    LOOP
        SELECT d.department_name, avg(e.salary)
        INTO deptname, avg_sal_dept
        FROM departments d
        INNER JOIN employees e
        ON d.department_id = e.department_id
        WHERE d.department_id = idx * 10
        GROUP BY d.department_name;

        dbms_output.put_line(deptname || ' ' || avg_sal_dept);
        idx := idx + 1;
    END LOOP;
END;
/
```

Loop Statements
FOR LOOP Statement

The **FOR** LOOP statement runs one or more statements while the **loop index** is in a **specified range**.

Without **REVERSE**:
- The value of index starts at **lower_bound** and **increases by one** with each iteration of the loop until it reaches **upper_bound**
- The statements **never run if** lower_bound is **greater than** upper_bound

With **REVERSE**:
- The value of index starts at **upper_bound** and **decreases by one** with each iteration of the loop until it reaches **lower_bound**
- If upper_bound is **less than** lower_bound, then the statements **never run**

An **EXIT**, **EXIT WHEN**, **CONTINUE**, or **CONTINUE WHEN** in the statements can cause the loop or the current iteration of the loop to end **early**.

```
-- FOR LOOP
[label] FOR IN [ REVERSE ] LOWER_BOUND .. UPPER_BOUND
LOOP
  statements
  [ EXIT [ WHEN condition ] ]
  [ CONTINUE [ WHEN condition ] ]
END LOOP [label];
```

```
DECLARE
    deptname departments.department_name%TYPE;
    avg_sal_dept number(7,2);
BEGIN
    FOR idx IN 1 .. 12
    LOOP
        dbms_output.put_line('Index =  ' || idx);
        IF idx = 12 THEN
            GOTO exit_for;
        END IF;

        SELECT d.department_name, avg(e.salary)
        INTO deptname, avg_sal_dept
        FROM departments d
            INNER JOIN employees e
                ON d.department_id = e.department_id
        WHERE d.department_id = idx * 10
        GROUP BY d.department_name;

        dbms_output.put_line(deptname || ' ' || avg_sal_dept);
    END LOOP;

    <<exit_for>>dbms_output.put_line('Done!');
END;
/
```

Conditional selection statements: **GOTO**

The sequential control statement **GOTO** transfers control to a label **unconditionally**.

Can be used to **exit** a LOOP. NULL statement only passes control to the next statement

```
GOTO label_example;

<<label_example>>
--Code
```

```
DECLARE
    deptname departments.department_name%TYPE;
    avg_sal_dept number(7,2);
BEGIN
    FOR idx IN 1 .. 12
    LOOP
        dbms_output.put_line('Index =  ' || idx);
        IF idx = 12 THEN
            GOTO exit_for;
        END IF;

        SELECT d.department_name, avg(e.salary)
        INTO deptname, avg_sal_dept
        FROM departments d
            INNER JOIN employees e
                ON d.department_id = e.department_id
        WHERE d.department_id = idx * 10
        GROUP BY d.department_name;

        dbms_output.put_line(deptname || ' ' || avg_sal_dept);
    END LOOP;

    <<exit_for>>dbms_output.put_line('Done!');
END;
/
```

Bibliography

Usha Krishnamurthy et al. *Oracle® Database: SQL Language Reference 19c*, Oracle Corporation, 2022 [pdf]

Usha Krishnamurthy et al. *Oracle® Database: SQL Language Reference 21c*, Oracle Corporation, 2022 [pdf]

Louise Morin et al. *Oracle® Database: Database PL/SQL Language Reference 19c*, Oracle Corporation, 2020 [pdf]

Louise Morin et al. *Oracle® Database: Database PL/SQL Language Reference 21c*, Oracle Corporation, 2021 [pdf]

Memes

- E: https://www.dailydot.com/unclick/lord-farquaad-e-meme/
- "What do the Number Mean" : COD
- Gandalf A Wizard Is Never Late : Lord of the Rings
- TLDR : https://www.seoreseller.com/wp-content/uploads/2014/01/tldr-cat.jpg
- Anya Pointer : https://www.instagram.com/programmer.meme/p/CysHW-4LgLL/
- Perhaps: https://en.meming.world/wiki/Perhaps
- Unlimited Power: Star Wars
- I made this : https://knowyourmeme.com/memes/i-made-this