Il linguaggio PL/SQL

Prof. Alessandra Lumini
alessandra.lumini@unibo.it

Per approfondimenti:

- ORACLE 11g Rel. 2 – Concepts
- ORACLE 11g Rel. 2 – PL/SQL Language Reference

Cos'è PL/SQL?

- Il linguaggio procedurale per l'estensione del linguaggio SQL di proprietà di Oracle
- Lo standard SQL è esteso dai principali sistemi commerciali:
  - Da Oracle con PL/SQL (e Java)
  - Da Access con Visual Basic
  - Da SQL Server con Transact-SQL

Procedurale vs dichiarativo?

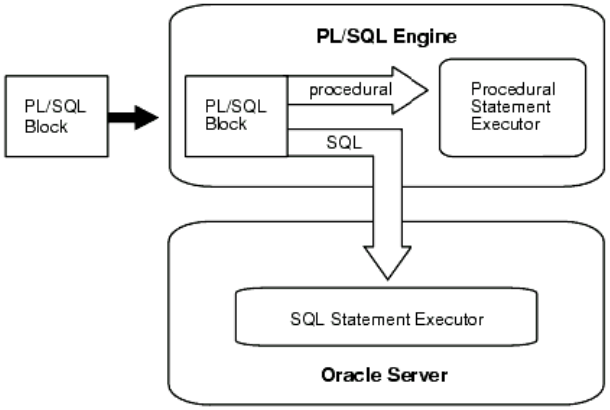
- Il linguaggio SQL è un linguaggio dichiarativo ottimale per inserire e reperire dati da un database ma non sufficientemente potente per codificare la logica applicativa
  - I programmi imperativi definiscono in modo esplicito un algoritmo per conseguire uno scopo
  - I programmi dichiarativi definiscono in modo esplicito soltanto lo scopo da raggiungere, lasciando che l'implementazione dell'algoritmo sia realizzata dal software di supporto. Il DBMS appunto!
- Un esempio non implementabile in SQL
  - Si vuole aumentare lo stipendio ai dipendenti che:
    - Non abbiano avuto più di 3 aumenti negli ultimi 5 anni
    - Il cui rendimento è superiore alla mediana degli altri dipendenti dello stesso dipartimento. Il rendimento è calcolato in base a un insieme di formule matematiche da usare in alternativa in base alle caratteristiche dell'impiegato
  - Nel caso in cui l'aumento sia applicabile
    - Va aggiornato lo stipendio dell'impiegato
    - Va compilato un report
    - Va inviata una mail al direttore del dipartimento e all'impiegato

Programmazione nei DBMS

- Server-side Programming: la logica di programmazione risiede all'interno del database mediante linguaggi supportati dal DBMS. Nel caso di Oracle: PL/SQL e Java
- Client-side Programming: i comandi SQL sono embedded nelle applicazioni sviluppate con un linguaggio procedurale (es. C++, Java)
  - Utilizzo di precompiler
  - Utilizzo di API (es. JDBC, OCI)
- La programmazione server-side pone molti vantaggi rispetto a quella client-side
  - Maggiori performance: minor quantità di dati trasferiti in rete
  - Minore quantità di memoria richiesta: una sola copia della procedura è caricata nella shared memory
  - Maggiore produttività: le procedure condivise da più applicazioni non devono essere replicate
  - Sicurezza: gli accessi alla procedura e dati sono regolati dai permessi degli utenti del db e non degli utenti dell'applicaizone

Il motore di PL/SQL

- Esegue le porzioni procedurali del codice ma invia al server oracle i comandi SQL



- Un blocco deve essere compilato prima che possa essere eseguito
  - Controllo sintattico
    - Struttura del comando, parole riservate e variabili
  - Binding
    - Controlla che gli oggetti referenziati esistano
  - Generazione del p-code
    - Istruzioni che il motore PL/SQL può eseguire

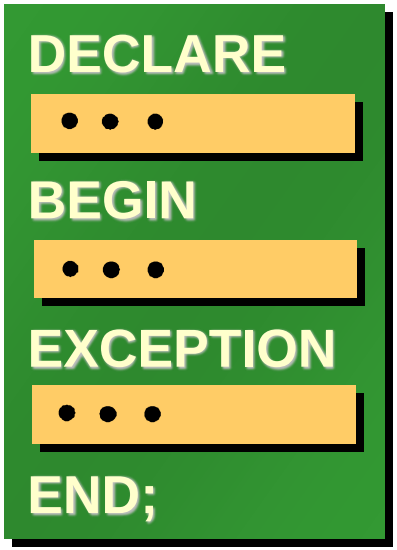
Blocchi PL/SQL

- I blocchi PL/SQL (Block)
  - Rappresentano l'unità elementare di codice PL/SQL
  - Normalmente contengono i comandi sufficienti a eseguire uno specifico compito
- Esistono due tipi di blocchi PL/SQL
  - Anonymous
  - Named: Si tratta di blocchi PL/SQL precompilati che vengono memorizzati nel database
    - stored procedure
    - function
    - trigger
    - package: gruppi di procedure e funzioni assemblate assieme tipicamente per affinità funzionale

Struttura di un blocco PL/SQL

- Sezione di dichiarazione
  - Per dichiarare, variabili, costanti, cursori,ecc.
  - E' opzionale
- Sezione di esecuzione
  - Descrive la logica dei comandi
  - Può contenere istruzioni SQL
  - E' obbligatoria
- Sezione di gestione delle eccezioni
  - Viene eseguita quando si presentano degli errori
  - E' opzionale

Attenzione nella definizione delle procedure e funzioni la clausola DECLARE è implicita



Esempio: anonymous

```
DECLARE
   qty_on_hand NUMBER(5);
BEGIN
   SELECT UNITSINSTOCK INTO qty_on_hand
      FROM NW_PRODUCTS
      WHERE PRODUCTNAME = 'Mozzarella di Giovanni' FOR UPDATE OF
UNITSINSTOCK;

   IF qty_on_hand > 0 THEN -- check quantity
      UPDATE NW_PRODUCTS SET UNITSINSTOCK = UNITSINSTOCK - 1
      WHERE PRODUCTNAME = 'Mozzarella di Giovanni';
   END IF;
   COMMIT;
END;
```

Procedure: definizione e call

- Una stored procedure è un blocco di codice PL/SQL dotato di un nome che viene mantenuto all'interno del database (procedure/funzioni)

```
CREATE OR REPLACE PROCEDURE nome_procedura
[(parametri)] IS
        Definizioni;
BEGIN
        Corpo procedura;
END;
```

```
CREATE FUNCTION nome_funzione [(parametri)] RETURN
tipo_dato IS
…
```

- Una procedura può essere richiamata utilizzando il comando call

```
CALL nome_procedura([parametri]);
```

Esempio: named

```
CREATE OR REPLACE PROCEDURE PROVA AS
   qty_on_hand NUMBER(5);
BEGIN
   SELECT UNITSINSTOCK INTO qty_on_hand
      FROM NW_PRODUCTS
      WHERE PRODUCTNAME = 'Mozzarella di Giovanni' FOR UPDATE OF
UNITSINSTOCK;

   IF qty_on_hand > 0 THEN -- check quantity
      UPDATE NW_PRODUCTS SET UNITSINSTOCK = UNITSINSTOCK - 1
      WHERE PRODUCTNAME = 'Mozzarella di Giovanni';
   END IF;
   COMMIT;
END;
```

Procedure: parametri

- Parametri è una sequenza di

```
[IN |OUT|IN OUT] <nome parametro> <tipo parametro>[,]
```

  che specifica eventuali valori passati in input
- TIPO_DATO non deve specificare lunghezza, precisione o scala.

  VARCHAR2(10) non è un tipo di dato valido VARCHAR2 si!
- Oracle deriva lunghezza, precisione o scala degli argomenti dall'ambiente da cui la procedura è chiamata.
- Di default i parametri sono utilizzati solo per il passaggio in ingresso delle informazioni (IN). Il passaggio IN OUT equivale a un passaggio di dati per riferimento in C.

DBMS_OUTPUT.PUT_LINE

- Mostra l'output a video
- In SQL Developer il risultato di un comando compare nella finestra Output DBMS
- DBMS_OUTPUT è un *package*
- .PUT_LINE è una *procedure* all'interno del package che stampa un'intera riga
- .PUT stampa una stringa (senza andare a capo)

```
BEGIN
   DBMS_OUTPUT.PUT_LINE ('ciao');
END;
```

Dichiarazione di una variabile

```
identifier [CONSTANT] datatype [NOT NULL]
                [:= | DEFAULT expr];
```

- La dichiarazione deve essere effettuata nella sezione DECLARE
- Di default le variabili sono inizializzate a NULL
- Le variabili sono dichiarate e inizializzate ogni volta che si accede al blocco
- Due variabili con lo stesso nome devono essere dichiarate in blocchi diversi
- Consigli:
  - Naming Conventions
    - Fino a 30 caratteri, non case sensitive, cominciano con una lettera e non possono contenere spazi
    - Non definire una variabile con il nome della colonna se queste vengono usate contemporaneamente. Utilizzare per esempio un prefisso per distinguerle (es. Quantity e vQuantity)
  - Dichiarate una variabile per riga

Assegnamento di un valore a una variabile

```
variablename := expression;
```

- Viene effettuato tramite comando di assegnamento nella sezione di esecuzione

```
CREATE PROCEDURE Esempio IS
   c_tax_rateCONSTANT NUMBER(3,2) := 8.25;
...
BEGIN
...
  v_hiredate := '31-DEC-98';
  v_fullname := ln || ', ' || fn;
...
```

- … oppure tramite il comando SELECT INTO

```
CREATE PROCEDURE Esempio IS
      v_max_len    number(7);
BEGIN
    SELECT max(length) INTO v_max_len
      FROM article;
...
```

La procedura «Hello world»

- Procedura per stampare a video la stringa «Hello world»

```
CREATE OR REPLACE PROCEDURE HELLOWORLD AS
v1 varchar2(12) :='Hello World!';
BEGIN
  DBMS_OUTPUT.PUT_LINE (v1);
END HELLOWORLD;
```

1. Scrivere una procedura che stampi in output la stringa BASI DI DATI AVANZATE come concatenazione di 4 variabili

4 tipi di variabili

- **Scalar**
  - Possono contenere un singolo valore
  - Corrispondono ai tipi di dati previsti per le tabelle Oracle più poche altre (es: Boolean)
- **Composite**
  - Permettono di manipolare gruppi di campi
  - es: una variabile di tipo %ROWTYPE memorizza un'intera riga
- **Reference**
  - Contengono puntatori
- **LOB** (Large OBjects)
  - Contengono elementi, chiamati *locators,* che specificano la posizione di oggetti di grosse dimensioni (es. immagini) che sono memorizzati separatamente

I principali tipi di dati scalari I

- **VARCHAR2 (lung. max.)**
  - Fino a 32,767 byte
- **CHAR [(lung. max.)]**
  - Fino a 32,767 byte
- **NUMBER [(precisione, scala)]**
  - precisione: 0-38
  - scala: -84 to 127
  - NUMBER(5,2) -> ddd.dd
- **DATE**
  - Da: January 1, 4712 BC A: December 31, 9999 AD
- **BOOLEAN**
  - TRUE o FALSE o NULL
  - Non ha nessun tipo corrispondente nei tipi degli attributi

```
CREATE PROCEDURE Esempio IS
    v_job               VARCHAR2(9);
    v_total_sal         NUMBER(9,2) := 0;
    v_duedate           DATE := SYSDATE + 7;
    v_valid             BOOLEAN NOT NULL := TRUE;
    c_tax_rate          CONSTANT NUMBER(3,2) := 8.25;
BEGIN …
```

Le variabili %TYPE

- Deve esserci corrispondenza tra il tipo di dati di una variabile PL/SQL e il rispettivo tipo della colonna nel DB
  - In caso contrario si verificherà un errore PL/SQL durante l'esecuzione
- Un tipo di dato "anchored" evita questo problema
  - indipendenza dei dati e adattamento runtime
- %TYPE dichiara una variabile in base a:
  - La definizione di una colonna del database
  - Un'altra variabile definita precedentemente
- Possibili prefissi per %TYPE sono:
  - I nomi della tabella e della colonna
  - Il nome della variabile precedentemente definita

```
CREATE PROCEDURE Esempio IS
  v_writerid     writer.writerid%TYPE;
  v_length       article.length%TYPE;
v_min_length     v_length%TYPE := 0;
BEGIN …
```

Lifetime

- Indica l'intervallo durante il quale una variabile esiste in memoria e può contenere un valore
- Lo spazio in memoria è **allocato** quando la variabile viene dichiarata
- Lo spazio in memoria è **deallocato** quando il programma raggiunge il comando END del blocco in cui è stata creata

Scope (Visibilità)

- La regione del programma in cui referenziare una variabile
- Le variabili dichiarate in una blocco PL/SQL sono **locali** al blocco e sono considerate **globali** per tutti i sotto blocchi
- La visibilità è inibita se nel blocco viene dichiarata una variabile con lo stesso nome.
  - Un blocco può fare riferimento a variabili dichiarate nei blocchi padre
  - Un blocco NON può fare riferimento a variabili dichiarate nei blocchi figli

Esempio

```
DECLARE
  v_sal           NUMBER(7,2) := 60000;
  v_comm          NUMBER(7,2) := v_sal * .20;
  v_message       VARCHAR2(255) := ' eligible for commission';
BEGIN
  DECLARE
    v_sal          NUMBER(7,2) := 50000;
    v_comm         NUMBER(7,2) := 0;
    v_total_comp   NUMBER(7,2) := v_sal + v_comm;
  BEGIN
    v_message := 'CLERK not'|| v_message;
  END;
  v_message := 'SALESMAN'|| v_message;
END;
```

- Determinare:
  - Il valore V_MESSAGE nel sottoblocco.
  - Il valore di V_TOTAL_COMP nel blocco principale.
  - Il valore di V_COMM nel sottoblocco.
  - Il valore di V_COMM nel blocco principale.
  - Il valore di V_MESSAGE nel blocco principale.

SELECT INTO

```
SELECT   select_list
INTO     {variable_name[, variable_name]...
         | record_name}
FROM     table
WHERE    condition;
```

- E’ necessario indicare ordinatamente il nome di una variabile per ogni colonna selezionata.
- L’interrogazione deve restituire una e una sola tupla
  - In caso contrario si genererà un errore
  - PL/SQL gestisce questi due errori generando due exception predefinite, che possono quindi essere trattate nella sezione EXCEPTION
    - NO_DATA_FOUND
    - TOO_MANY_ROWS

Query con una singola riga

- Procedura per stampare a video il numero di prodotti

```
CREATE OR REPLACE PROCEDURE NUMPRODOTTI AS
nProd NUMBER(5,0);
BEGIN
  SELECT count(*) INTO nProd FROM NW.PRODUCTS ;
  DBMS_OUTPUT.PUT_LINE ('Numero di Prodotti:' || nProd);
END;
```

2. Scrivere una procedura che stampi il numero di ordini e il ricavo totale

SQL statico in PL/SQL

- Con il termine *SQL statico* si identificano i comandi SQL direttamente inclusi nel codice PL/SQL e sottoposti al processo di compilazione
- In SQL statico:
  - DDL non è supportato
    - CREATE TABLE, CREATE INDEX, ALTER TABLE, DROP VIEW
  - DCL non è supportato
    - GRANT, REVOKE, CREATE USER, DROP ROLE, ALTER USER
  - DML è supportato
    - INSERT, UPDATE, DELETE
  - TCL è supportato
    - COMMIT, ROLLBACK, SAVEPOINT

Esempio

```
DECLARE
  v_sal_increase   emp.sal%TYPE := 2000;
BEGIN
  UPDATE  emp
  SET        sal = sal + v_sal_increase
  WHERE    job = 'ANALYST';
  COMMIT;
END;
```

```
CREATE OR REPLACE PROCEDURE Esempio IS
  v_deptno   emp.deptno%TYPE := 10;

BEGIN
  DELETE FROM    emp
  WHERE          deptno = v_deptno;

  COMMIT;
END;
```

SQL Dinamico

- La creazione dinamica di comandi SQL all'interno di un blocco di codice PL/SQL può essere utile quando:
  - Il comando SQL non è noto a compile time
  - Il comando SQL non è supportato come SQL statico
    - GRANT, REVOKE, CREATE USER, DROP ROLE, ALTER USER
- Dove l'SQL dinamico non è necessario, l'SQL statico è preferibile perché la compilazione verifica la correttezza sintattica del comando, e degli oggetti che esso referenzia
- Esistono più metodi per costruire un comando di SQL dinamico:
  - EXECUTE IMMEDIATE
  - OPEN FOR, FETCH, CLOSE
  - DBMS_SQL Package

EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE [dynamic SQL string statement without terminator]
[INTO {define_variable [, define_variable] ... | record}]
[USING [IN|OUT|IN OUT] bind_argument [, [IN|OUT|IN OUT] bind_arguments] ]
```

- Il comando può non richiedere parametri di input/output
  - In questo caso INTO e USING non sono necessari
- La *stringa SQL* non deve terminare con ;
- Se il comando è un SELECT che restituisce una sola tupla si può utilizzare
  - INTO per specificare le variabili di output
  - USING per specificare le variabili di input e output
- La stringa SQL può contenere placeholder per argomenti di binding, ma tali argomenti non possono essere utilizzati per passare i nomi degli oggetti dello schema (tabelle o colonne). Si possono passare interi, date e stringhe ma non booleani o valori nulli
- Negli altri casi è necessario comporre la stringa SQL

Esempio EXECUTE IMMEDIATE

```
CREATE OR REPLACE FUNCTION get_row_cnts(p_tname in varchar2)
RETURN number AS
l_cnt number;
BEGIN
  EXECUTE IMMEDIATE 'select count(*) from ' ||p_tname INTO l_cnt;
  RETURN l_cnt;
END;
```

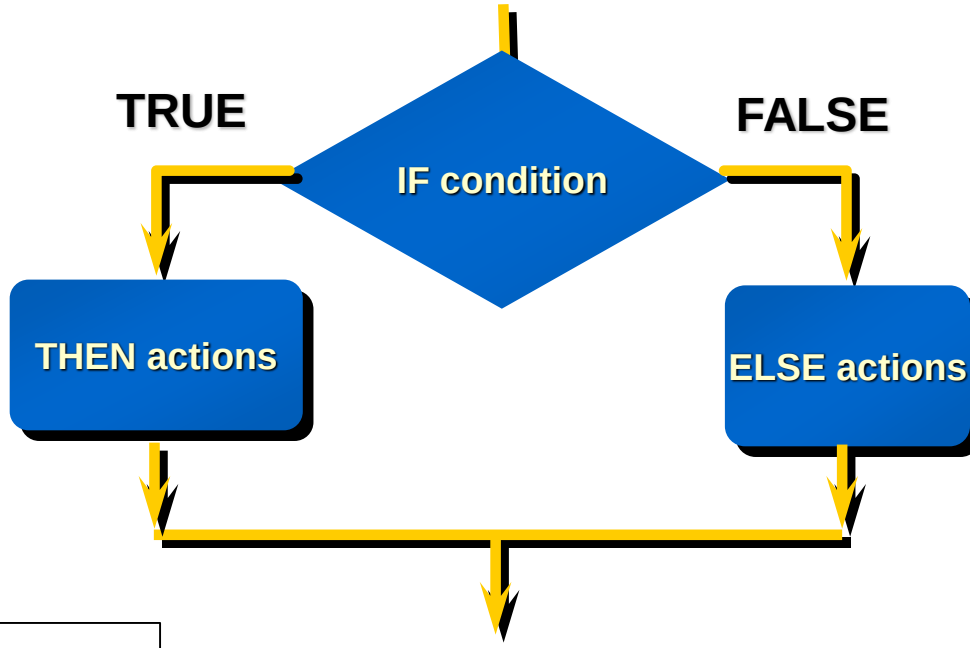
```
CREATE OR REPLACE FUNCTION get_num_of_employees (p_loc VARCHAR2,
p_job VARCHAR2) RETURN NUMBER IS
v_query_str VARCHAR2(1000);
v_num_of_employees NUMBER;
BEGIN
 v_query_str := 'SELECT COUNT(*) FROM emp_' || p_loc || `
 WHERE job = :bind_job' ;
 EXECUTE IMMEDIATE v_query_str INTO v_num_of_employees
 USING p_job;
 RETURN v_num_of_employees;
END;
```

Controllo del flusso di elaborazione

- Per cambiare il flusso di esecuzione all'interno di un blocco di codice sono disponibili i seguenti comandi
  - IF-THEN
    - Seleziona se eseguire o non un comando
  - IF-THEN-ELSE
    - Seleziona quale di due comandi debbano essere eseguiti in mutua esclusione
  - IF-THEN-ELSIF
    - Seleziona quale di più comandi debbano essere eseguiti in mutua esclusione
- Attenzione:
  - ELSIF è una parola
  - END IF sono due parole

IF-THEN-ELSE

```
IF condition THEN
  statement(s);
ELSE
  statement(s);
END IF;
```




```
...
IF sales > quota THEN
   bonus:=compute_bonus(empid);
   UPDATE payroll
     SET pay = pay + bonus
     WHERE empno = emp_id;
END IF;
...
```

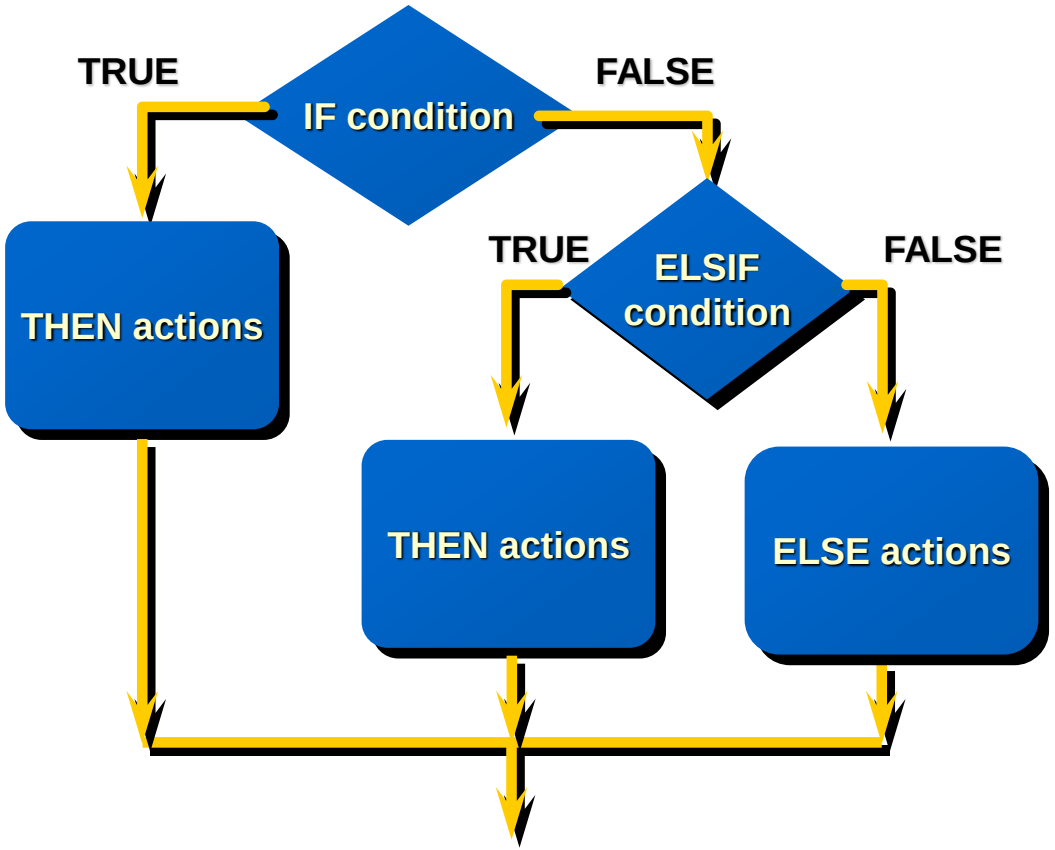
Esempio

```
CREATE OR REPLACE PROCEDURE Esempio IS
   qty_on_hand NUMBER(5);
BEGIN
   SELECT quantity INTO qty_on_hand
      FROM inventory
      WHERE product = 'TENNIS RACKET' FOR UPDATE OF quantity;

   IF qty_on_hand > 0 THEN -- check quantity
      UPDATE inventory SET quantity = quantity - 1
         WHERE product = 'TENNIS RACKET';
      INSERT INTO purchase_record
         VALUES ('Tennis racket purchased', SYSDATE);
   ELSE
      INSERT INTO purchase_record
        VALUES ('Out of tennis rackets', SYSDATE);
   END IF;
   COMMIT;
END;
```

IF-THEN-ELSIF

- Simile a un comando SWITCH
- Quando è possibile si usi ELSIF invece che un insieme di IF in cascata
- Il codice risultante sarà più leggibile
- Le condizioni devono essere mutualmente esclusive
- Dopo l'esecuzione il flusso viene ripreso al primo comando che segue END IF



IF-THEN-ELSIF Esempio

```
CREATE OR REPLACE PROCEDURE Esempio IS
   v_title          article.title%TYPE;
   v_length         article.length%TYPE;
   v_descr          VARCHAR2(6);
BEGIN
   SELECT title, length INTO v_title, v_length
      FROM article
      WHERE articlenum = &sv_articlenum;
   IF v_length <=1500 THEN
      v_descr := 'Brief';
   ELSIF v_length BETWEEN 1501 and 2500 THEN
      v_descr := 'Short';
   ELSIF v_length BETWEEN 2501 and 4000 THEN
      v_descr := 'Medium';
   ELSE
      v_descr := 'Long';
   END IF;
   DBMS_OUTPUT.PUT_LINE('Article ' || v_title || ' is ' ||
v_descr);
END;
```

Istruzione CASE

```
CREATE OR REPLACE PROCEDURE Esempio IS
   v_title          article.title%TYPE;
   v_length         article.length%TYPE;
   v_descr          VARCHAR2(6);
BEGIN
   SELECT title, length INTO v_title, v_length
      FROM article
      WHERE articlenum = &sv_articlenum;
CASE
   WHEN v_length <=1500 THEN
      v_descr := 'Brief';
   WHEN v_length BETWEEN 1501 and 2500 THEN
      v_descr := 'Short';
   WHEN v_length BETWEEN 2501 and 4000 THEN
      v_descr := 'Medium';
   ELSE
      v_descr := 'Long';
END;
DBMS_OUTPUT.PUT_LINE('Article ' || v_title || ' is ' || v_descr);
END;
```

Condizioni complesse

- I valori null sono gestiti tramite l'operatore IS NULL
  - es: `IF v_gender IS NULL THEN`
- Qualsiasi espressione aritmetica che comprenda un NULL comporta il risultato NULL
- Nella concatenazione di più variabili la presenza di un NULL viene trattata come una stringa vuota
- Condizioni complesse vengono create utilizzando gli operatori logici NOT, AND, and OR
  - es:
```
IF v_length > 500 AND v_type = 'BUS' THEN ...
```
  - es:
```
IF v_length > 500 OR  v_type = 'BUS' THEN ...
```
  - es:
```
IF v_length > 500 OR  v_type = 'BUS'  AND
      v_type = 'LAW' THEN ...
```
  - La precedenza tra gli operatori è così fissata: NOT, AND, OR

Logica a tre valori

- Le istruzioni di controllo del flusso gestiscono anche predicati in cui sono coinvolte variabili con valori NULL.

AND	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

- Si può verificare esplicitamente se una variabile ha valore NULL mediante gli operatori IS [NOT] NULL

Esercizi: Uso di condizioni

- Procedura per stampare il numero di ordini N gestiti da un impiegato (in input). Se N>100 stampa «high», se minore di 50 stampa «low» altrimenti «medium»

```
create or replace PROCEDURE ProdImp(idImp number) AS
nOrd NUMBER(5,0);
BEGIN
  SELECT count(*) into nOrd FROM NW.ORDERS WHERE EmployeeID=idImp ;
  IF (nOrd) > 100 THEN
        DBMS_OUTPUT.PUT_LINE ('High:' || nOrd);
  ELSIF (nOrd) < 50 THEN
        DBMS_OUTPUT.PUT_LINE ('Low:' || nOrd);
  ELSE
        DBMS_OUTPUT.PUT_LINE ('Medium:' || nOrd);
  END IF;
END;
```

3. Scrivere una procedura che classifica un cliente in base al totale acquistato (best >100K /standard/worst <5K)

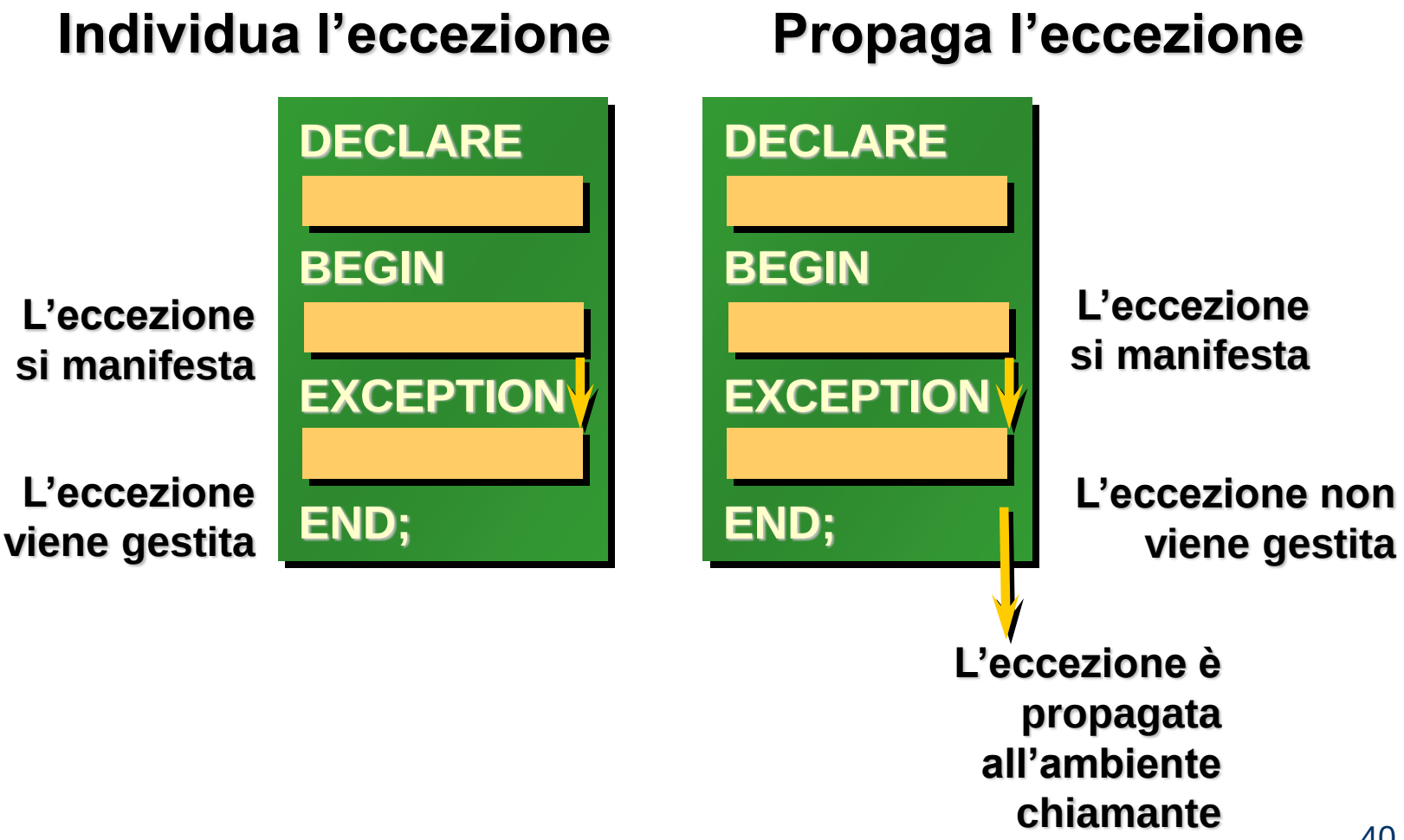
Il concetto di eccezione I

- Cosa è una exception?
  - Un identificatore PL/SQL che viene valorizzato durante l'esecuzione di un blocco
  - L'esecuzione viene trasferita al corrispondente gestore dell'eccezione nella sezione exception del blocco
- Come avviene la valorizzazione?
  - Automaticamente (implicitamente) quando si verifica un errore runtime
  - Esplicitamente se nel codice è presente l'istruzione RAISE
- Come vengono gestite?
  - Includendo una routine corrispondente nella sezione exception
- Cosa avviene in caso contrario?
  - Il blocco PL/SQL termina con un errore
  - L'eccezione è propagata all'applicazione chiamante
  - SQL*Plus mostra il corrispondente messaggio di errore

Il concetto di eccezione II

- Possono essere definiti molti tipi di eccezioni ognuno associato a un proprio insieme di comandi
  - Ogni gestore è identificato da una clausola WHEN, che specifica una o più eccezioni, seguita da un insieme di comandi
- Si può verificare una sola eccezione per volta
- Il gestore **OTHERS**
  - Controlla ogni eccezione non trattata esplicitamente
  - Deve essere l'ultima eccezione nella lista
- Le eccezioni possono essere:
  - Internally defined: vengono attivate dal sistema automaticamente e sono associate a un codice di errore
  - Predefined: vengono attivate dal sistema automaticamente e sono associate a un codice di errore e a un nome
  - User-defined: sono definite e attivate da un utente tramite il comando RAISE

Gestione delle eccezioni



Eccezioni predefinite

- PL/SQL predefinisce alcune eccezioni comuni:
  - NO_DATA_FOUND (ORA-01403)
    - Una SELECT INTO ha restituito 0 righe
  - TOO_MANY_ROWS (ORA-01422)
    - Una SELECT INTO ha restituito più di una riga
  - VALUE_ERROR (ORA-06502)
    - Si è verificato un errore aritmetico, numerico, di conversione o su un vincolo (es: si è tentato di assegnare il valore NULL a una variabile NOT NULL)
  - ZERO_DIVIDE (ORA-01476)
  - DUP_VAL_ON_INDEX (ORA-00001)
- Ignorare l'eccezione
  - WHEN <name> THEN NULL;

```
BEGIN ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement;

  WHEN TOO_MANY_ROWS THEN
    statement;

  WHEN OTHERS THEN
    statement;
END;
```

NO_DATA_FOUND

- Le funzioni di aggregazione SQL (es. AVG, SUM) restituiscono sempre un valore o NULL
- Un comando SELECT INTO che nella select list include solo funzioni di aggregazione non attiva mai l'eccezione NO_DATA_FOUND.
- Ovviamente ciò non è vero se il comando SELECT INTO prevede anche un raggruppamento

```
CREATE OR REPLACE PROCEDURE Esempio(v_id varchar2) IS
  v_name VARCHAR2(50);
BEGIN
  SELECT fn || ' ' || ln INTO v_name FROM writer
  WHERE writerid = v_id;
  DBMS_OUTPUT.PUT_LINE('Writer '||v_id||' is '||v_name);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No such writer: '|| v_id);
END;
```

Esercizi: Gestione delle eccezioni

- Stampa dei dati di un cliente dato il nome. Gestire l'assenza del cliente o la presenza di più di un record

```
CREATE OR REPLACE PrintCliente(Nome VARCHAR2) AS
vCliente NW_CUSTOMERS%ROWTYPE;
BEGIN
  SELECT * into vCliente FROM NW.CUSTOMERS WHERE COMPANYNAME
LIKE Nome;
  DBMS_OUTPUT.PUT_LINE ('Cliente: ' || vCliente.COMPANYNAME
|| ' ID: ' || vCliente.CUSTOMERID );
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('Cliente non trovato');
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE ('Nome cliente non univoco');
END;
```

4. Scrivere una procedura che calcoli il totale degli ordini effettuati in una certa data. Gestire l'assenza di ordini.

Cicli

- PL/SQL mette a disposizioni 4 istruzioni per il controllo dei cicli:
  - Cicli semplici
  - Ciclo WHILE
  - Cicli FOR numerici
  - Cicli FOR per cursori

Cicli semplici

- Il comando EXIT determina l'uscita incondizionata dal ciclo

```
CREATE PROCEDURE Esempio_PreTest(p_end_at NUMBER) IS
    v_counter    NUMBER(2) := 1;
BEGIN
  LOOP
    EXIT WHEN v_counter > p_end_at;
    DBMS_OUTPUT.PUT_LINE(v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

```
CREATE PROCEDURE Esempio_PostTest(p_end_at NUMBER) IS
    v_counter    NUMBER(2) := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE(v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > p_end_at;
  END LOOP;
END;
```

Cicli WHILE

- Ripete i comandi finché la condizione è TRUE
  - Pre-Test: la condizione viene verificata prima di eseguire i comandi
  - Il ciclo termina quando la condizione diviene FALSE o NULL
  - Può essere utilizzato il comando EXIT per terminare in maniera anticipata il ciclo

```
WHILE condition1
LOOP
   statement1;
   statement2;
   [EXIT WHEN cond2]
END LOOP;
```

```
CREATE PROCEDURE Esempio_WhileTest(p_end_at NUMBER) IS
    v_counter    NUMBER(2) := 1;
BEGIN
  WHILE v_counter < p_end_at;
  LOOP
    DBMS_OUTPUT.PUT_LINE(v_counter);
    v_counter := v_counter + 1;
END LOOP;
END;
```

Cicli FOR numerici

- Un contatore implicito viene incrementato ad ogni ciclo
  - L'incremento è automatico ed è sempre di 1
- Il ciclo continua finché il contatore è ≤ upper
- Se lower > upper comandi non vengono eseguiti
- *lower* e *upper* possono essere numeri, variabili, o espressioni che possano essere sempre valutati come interi
- Il *counter* è definito e può essere referenziato solo all'interno del ciclo
- Può essere utilizzato il comando EXIT per terminare in maniera anticipata il ciclo

```
FOR counter IN [REVERSE] lower..upper
LOOP
   statement1;
   statement2;
   [IF cond EXIT]
   . . .
END LOOP;
```

Esercizi: uso di cicli

- Stampa N date a partire da una data iniziale

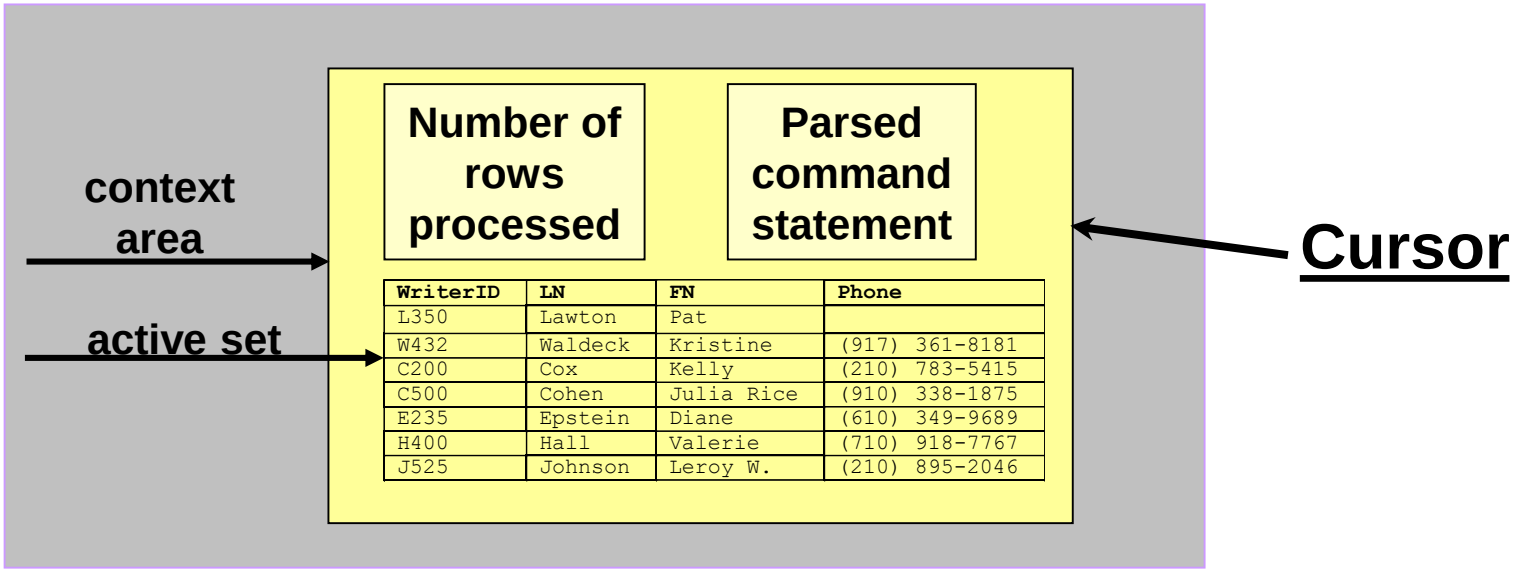
```
CREATE OR REPLACE PROCEDURE PrintDates(iDate DATE, nPrint NUMBER) IS
    v_counter    NUMBER(2) := 0;
BEGIN
  LOOP
    EXIT WHEN v_counter >= nPrint;
 DBMS_OUTPUT.PUT_LINE(TO_CHAR(iDate+v_counter,'DD FMMonth YYYY'));
    v_counter := v_counter + 1;
  END LOOP;
END;

--oppure
for v_counter in 1 .. nPrint
LOOP
DBMS_OUTPUT.PUT_LINE(TO_CHAR(iDate+v_counter,'DD FMMonth YYYY'));
END LOOP;
```

5. Scrivere una procedura che stampa il totale di N ordini a partire da IDOrd (N e IDOrd in input). Gestire con una eccezione l'assenza di uno o più ordini

Il concetto di cursore

- Ogniqualvolta si sottoponga al sistema un comando SQL, Oracle alloca un'area di memoria in cui il comando viene analizzato ed eseguito. Tale area è detta ***context area***.
- Un ***cursore*** è un puntatore alla locazione di memoria di una context area
- Ogni comando SQL eseguito da Oracle ha associato un proprio cursore



Due tipi di Cursori

- Cursori Impliciti
  - Il server Oracle apre implicitamente un cursore durante l'esecuzione di un comando DML o di ogni query PL/SQL SELECT INTO
  - Il cursore è gestito automaticamente
    - Non si può utilizzare OPEN, FETCH, CLOSE per controllarlo
  - PL/SQL fa riferimento al più recente cursore implicito come cursore **SQL**
- Cursori Espliciti
  - Sono dichiarati e gestiti direttamente dal codice
  - Sono utilizzati per elaborare le singole righe restituite da un comando SQL multiple-row
  - Puntano alla riga corrente nell' *active set*

Attributi dei cursori impliciti

- E' possibile utilizzare gli attributi del cursore sql per verificare il risultato di un comando SQL

SQL%ROWCOUNT	**Numero di righe coinvolte dal più recente comando SQL**
SQL%FOUND	**Attributo Boolean che è TRUE se l'ultimo comando SQL ha restituito almeno una riga**
SQL%NOTFOUND	**Attributo Booleano che è TRUE se l'ultimo comando SQL non ha restituito nemmeno una riga**
SQL%ISOPEN	**E' sempre FALSE poiché PL/SQL chiude i cursori impliciti immediatamente dopo l'esecuzione**

Esempio

- Dai a ogni scrittore freelance un aumento del 25% e mostra il numero di righe modificate.

```
CREATE PROCEDURE Esempio IS
-- nessun cursore è dichiarato esplicitamente
BEGIN
   UPDATE writer
      SET amount = amount * 1.25
    WHERE freelancer = 'Y';
  DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT ||' rows changed.');
 COMMIT;
END;
```

```
SQL> /
6 rows changed.

PL/SQL procedure successfully completed.
```

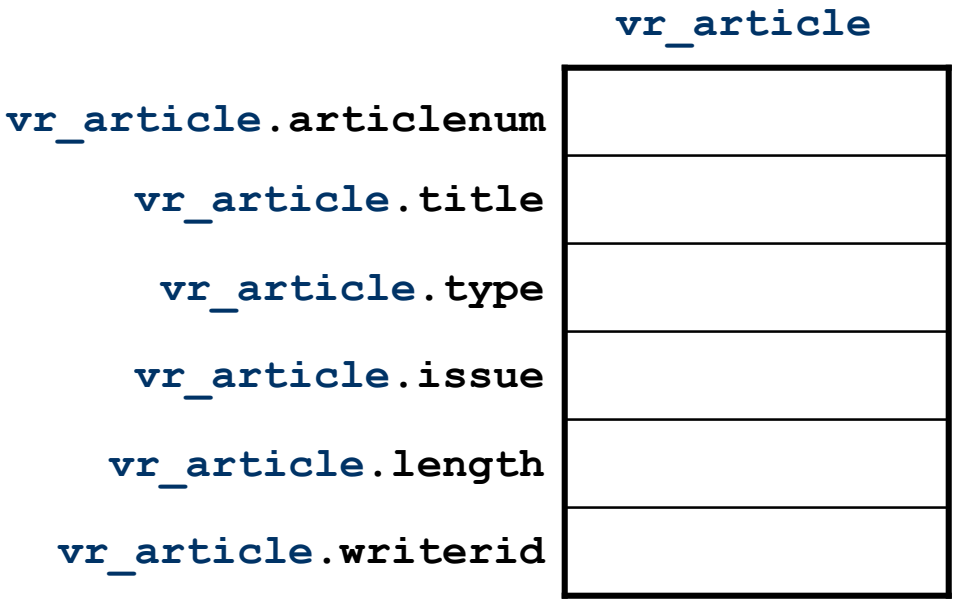
Record PL/SQL

- Un record PL/SQL è un gruppo di attributi correlati memorizzati in una tabella, ognuno col proprio nome e tipo
- Un record PL/SQL è quindi un tipo composto in cui i singoli campi sono trattati come un'unità logica
- Sono convenienti per gestire le righe dell' active set, poiché permettono di eseguire il FETCH di un'intera riga.
  - I valori della riga vengono caricati nei campi corrispondenti
- Il tipo %ROWTYPE permette di dichiarare una variabile di tipo *record* basandosi su un insieme di campi appartenenti a una tabella, vista o cursore.
- E' necessario anteporre a %ROWTYPE il nome della tabella, vista o cursore a cui il record è associato.

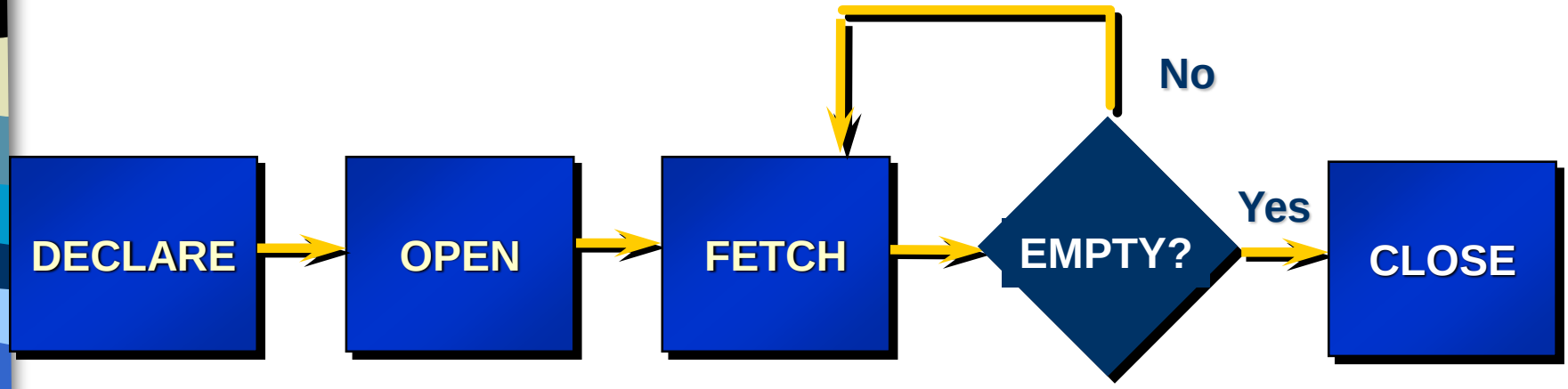
```
DECLARE
   vr_article  article%ROWTYPE;
   . . .
```

%ROWTYPE

- Ci si riferisce a un membro di un campo utilizzando la sintassi
  - recordvariable_name.fieldname
- I campi senza un valore iniziale sono inizializzati a NULL.
- Il tipo e il numero delle colonne nel database può cambiare.



Controllo di un cursore esplicito



Crea la context area

Definisce la query che produce i dati

Esegue la query

Identifica l'active set

Posiziona il cursore *prima* della riga iniziale

Carica la riga corrente nelle variabili

Fa avanzare il cursore alla prossima riga

Controlla l'esistenza di ulteriori righe

Esegue nuovamente il FETCH se è stata trovata una riga

Rilascia l'active set

Disabilita il cursore e rilascia le risorse

Dichiarazione di un cursore esplicito

```
CURSOR cursor_name IS
     select_statement;
```

- *select_statement* è un qualsiasi comando SELECT
  - Può includere join, operatori di set e subquery
  - Se è necessario processare le righe in una determinata sequenza si può utilizzare la clausola ORDER BY nella query.
- E' possibile fare riferimento a variabili all'interno della query, ma queste devono essere definite anticipatamente.

Apertura di un cursore

```
OPEN  cursor_name;
```

- Esegue l'interrogazione e identifica l'active set.
- Posiziona il puntatore *prima* della prima riga nell'active set.
  - Le righe non vengono caricate nelle variabili fino all'esecuzione del comando FETCH
- Non si verifica alcuna eccezione se la query non restituisce valori.

Attributi dei Cursori espliciti

- Permettono di ottenere informazioni sui cursori espliciti

Attributo	Tipo	Descrizione
%ISOPEN	Boolean	Restituisce TRUE se il cursore è open
%NOTFOUND	Boolean	Restituisce TRUE se il FETCH più recente non ha restituito righe
%FOUND	Boolean	Restituisce TRUE se il FETCH più recente ha restituito righe.
%ROWCOUNT	Number	Restituisce il numero totale di righe restituite (ossia fetched)

Leggere i dati dal cursore

```
FETCH cursor_name INTO [variable1, variable2, ...]
                       | record_name];
```

- I dati possono essere inseriti in un record o in un insieme di variabili
- Dopo un FETCH, il cursore avanza alla prossima riga dell'active set
- Dopo ogni FETCH è necessario verificare se il cursore contiene delle righe
  - Se un cursore non acquisisce valori l'active set è stato completamente elaborato
  - Non vengono create delle eccezioni
  - Le variabili/record mantengono i valori precedenti
- La lettura dei record può essere inserita in un ciclo loop in cui la condizione di uscita dipende dallo stato del cursore

```
EXIT WHEN cursor_name%NOTFOUND
```

Chiusura di un Cursore

```
CLOSE     cursor_name;
```

- Chiude il cursore dopo aver completato l'elaborazione.
- Disabilita il cursore rendendo indefinito l'active set.
- Non è possibile eseguire FETCH su un cursore chiuso.
  - Provocherebbe una eccezione di tipo INVALID_CURSOR
  - La riapertura del cursore provocherà la riesecuzione dell'interrogazione

Esempio

- Caricamento dei dati dei cursori in variabili **PL/SQL**

```
CREATE PROCEDURE Esempio IS
   CURSOR writer_cursor IS
      SELECT ln, phone
      FROM   writer
      ORDER BY ln;
   v_ln     writer.ln%TYPE;
   v_phone  writer.phone%TYPE;

BEGIN
   OPEN writer_cursor;
   LOOP
      FETCH writer_cursor INTO v_ln, v_phone;
      EXIT WHEN writer_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RPAD(v_ln,40) || v_phone);
   END LOOP;
   CLOSE writer_cursor;
END;
```

Esempio

- Caricamento dei dati dei cursori in record **PL/SQL**

```
CREATE PROCEDURE Esempio IS
   CURSOR writer_cursor IS
      SELECT ln, phone
      FROM   writer
      ORDER BY ln;
   v_rec     writer_cursor%ROWTYPE;

BEGIN
   OPEN writer_cursor;
   LOOP
      FETCH writer_cursor INTO v_rec;
      EXIT WHEN writer_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RPAD(v_rec.ln,40) || v_rec.phone);
   END LOOP;
   CLOSE writer_cursor;
END;
```

Cicli FOR e cursori

```
FOR record_name IN cursor_name LOOP
   statement1;
   statement2;
   . . .
END LOOP;
```

- Semplifica l'utilizzo di cursori espliciti
  - Il cursore è aperto e ne viene recuperata una riga per ogni iterazione; il cursore è chiuso automaticamente dopo l'elaborazione dell'ultima riga.
  - Il record PL/SQL che conterrà i dati viene definito automaticamente
  - Le operazioni di OPEN, FETCH, e CLOSE avvengono automaticamente

ESEMPIO

- Recupera nome e cognome di ogni scrittore

```
CREATE PROCEDURE Esempio IS
   CURSOR writer_cursor IS
      SELECT ln, phone
      FROM   writer
      ORDER BY ln;
BEGIN
  FOR vr_writer IN writer_cursor LOOP  -- implicit open/fetch
     DBMS_OUTPUT.PUT_LINE(RPAD(vr_writer.ln,40) ||
                                vr_writer.phone);
  END LOOP; -- Chiusura implicita
END;
```

- Si noti la riduzione nel numero dei comandi
  - Nessuna dichiarazione per vr_writer
  - Esecuzione automatica di OPEN, FETCH e CLOSE

ESEMPIO

- Con cursore implicito

```
CREATE PROCEDURE Esempio IS
BEGIN
  FOR vr_writer IN (SELECT ln, phone FROM   writer ORDER BY ln) LOOP  --
implicit open/fetch
     DBMS_OUTPUT.PUT_LINE(RPAD(vr_writer.ln,40) || vr_writer.phone);
  END LOOP; -- Chiusura implicita
END;
```

- Ancora più compatto
  - Nessuna dichiarazione per il cursore
  - Meno leggibile

I cursori – ciclo FOR

- Definire un cursore per visualizzare le categorie

```
CREATE PROCEDURE PrintCategories IS
CURSOR cCat IS
SELECT CATEGORYID, CATEGORYNAME, DESCRIPTION FROM NW.CATEGORIES;

BEGIN
  FOR vCat IN cCat
  LOOP -- implicit open/fetch
    DBMS_OUTPUT.PUT_LINE(vCat.CATEGORYNAME || ': ' ||
        vCat.DESCRIPTION);
  END LOOP; -- Chiusura implicita
END;
```

6. Definire un cursore per stampare i corrieri e quanti ordini hanno gestito

I cursori – FETCH esplicito

- Definire un cursore per visualizzare le categorie

```
CREATE PROCEDURE PrintCategories IS
CURSOR cCat IS
SELECT CATEGORYID, CATEGORYNAME, DESCRIPTION FROM NW.CATEGORIES ;

vCat cCat%ROWTYPE;
BEGIN
OPEN cCat;
   LOOP
      FETCH cCat INTO vCat;
      EXIT WHEN cCat%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(vCat.CATEGORYNAME || ':  '  ||
vCat.DESCRIPTION);
   END LOOP;
   CLOSE cCat;
END;
```

7. Definire un cursore per stampare i 10 clienti più affezionati (per numero di ordini)

Cursori con Parametri

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

- I parametri permettono di passare al cursore dei valori utilizzati nella query che carica i dati durante l'apertura.
- Un cursore può essere aperto più volte nello stesso blocco producendo active set diversi

```
OPEN cursor_name(par_val);
```

- Oppure:

```
FOR record_name IN cursor_name(par_val) LOOP
  statement1;
. . .
END LOOP;
```

Esempio

```
CREATE PROCEDURE Esempio IS
  CURSOR writer_cursor (p_flstatus IN writer.freelancer%TYPE)
IS
    SELECT      ln, phone
    FROM        writer
    WHERE       freelancer = p_flstatus;
  vr_writer  writer_cursor%ROWTYPE;

BEGIN
  OPEN writer_cursor('Y');
  LOOP
      FETCH writer_cursor INTO vr_writer;
      EXIT WHEN writer_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(RPAD(vr_writer.ln,40) ||
                           vr_writer.phone);
   END LOOP;
   CLOSE writer_cursor;
END;
```

Esercizio

- Stampa di tutti i prodotti di una categoria (in input)

```
CREATE PROCEDURE PrintProd (vIDCat number) IS
   CURSOR cProd (pCat IN NW.CATEGORIES.CATEGORYID%TYPE) IS
   SELECT *  FROM NW.PRODUCTS WHERE  CATEGORYID=pCat;
BEGIN
FOR vProd IN cProd(vIDCat) LOOP
     DBMS_OUTPUT.PUT_LINE(vProd.PRODUCTID || ':  '  ||
vProd.PRODUCTNAME);
  END LOOP;
END;
```

8. Data una città in input, stampare tutti i clienti residenti e per ciascuno la lista dei prodotti ordinati.

FOR UPDATE

```
SELECT ...
FROM   ...
FOR UPDATE [OF column_reference][NOWAIT];
```

- Applica un lock alle righe selezionate dal cursore in modo che sia possibile modificare o cancellare i valori all'interno del codice
- Il lock è applicato al momento dell'apertura del cursore non durante la fase di fetch
- Il lock è rilasciato al momento del COMMIT o ROLLBACK da eseguire al termine del ciclo
  - L'esecuzione di COMMIT o ROLLBACK per ogni riga provoca errore (ORA-01002)
- Se il cursore applica una selezione su più tabelle tramite FOR UPDATE è possibile limitare il lock a una sola tabella. Il lock è applicato solo alle righe delle tabelle di cui è citato un campo nella clausola FOR UPDATE.
- La clausola FOR UPDATE è l'ultima di ogni query di SELECT.

FOR UPDATE

```
SELECT ...
FROM   ...
FOR UPDATE [OF column_reference][NOWAIT];
```

- NOWAIT indica al server di non attendere se sulle tabelle è attivo un lock di un'altra sessione.
  - Si verifica una exception
  - Il controllo è restituito al programma che può eseguire altre operazioni prima di tentare di riacquisire il lock

FOR UPDATE Esempio

```
CREATE PROCEDURE Esempio IS
   CURSOR c_stud_zip IS
      SELECT s.student_id, z.city
        FROM student s, zipcode z
        WHERE z.city = 'Brooklyn'
        AND   s.zip = z.zip
        FOR UPDATE OF s.phone;
BEGIN
   FOR r_stud_zip IN c_stud_zip
   LOOP
       UPDATE student
          SET phone = '333'|| substr(phone,4)
          WHERE student_id = r_stud_zip.student_id;
   END LOOP;
   COMMIT;
END;
```

Cosa viene bloccato? Quando?

Il COMMIT è eseguito alla fine

WHERE CURRENT OF

```
WHERE CURRENT OF cursor ;
```

- Referenzia la riga corrente di un cursore esplicito.
- Permette di eseguire UPDATE o DELETE della riga corrente utilizzando una clausola WHERE semplificata.
  - Non richiede di creare la condizione che specifichi a quale riga applicare l'operazione poiché questa viene applicata alla riga corrente.
- E' necessario utilizzare FOR UPDATE nella definizione del cursore in modo da applicare un lock sulla tabella
  - In caso contrario si verificherà un errore

Esempio

```
CREATE PROCEDURE Esempio IS
    CURSOR c_stud_zip IS
       SELECT s.student_id, z.city
         FROM student s, zipcode z
         WHERE z.city = 'Brooklyn'
         AND   s.zip = z.zip
         FOR UPDATE OF phone;
BEGIN
     FOR r_stud_zip IN c_stud_zip
     LOOP
        UPDATE student
           SET phone = '718'|| substr(phone,4)
           WHERE CURRENT OF c_stud_zip;
     END LOOP;
     COMMIT;
END;
```

Esercizio

- Aumenta del 10% il prezzo dei prodotti

```
create table NW1_Products as select *  from NW.Products;

CREATE PROCEDURE IncPrice  IS
   CURSOR cProd  IS
   SELECT *  FROM NW1_PRODUCTS FOR UPDATE OF UNITPRICE;
BEGIN
FOR vProd IN cProd LOOP
     UPDATE NW1_PRODUCTS SET UNITPRICE=1.1*UNITPRICE
     WHERE CURRENT OF cProd;
END LOOP;
COMMIT;
END;

CREATE PROCEDURE IncPrice1  IS
BEGIN
  UPDATE NW1_PRODUCTS SET UNITPRICE=1.1*UNITPRICE;
END;
```

9. Aumenta del P% il prezzo dei prodotti di un fornitore F, se il prodotto è già in riordino l'aumento sarà del P/2%

Gli array

- Un VARRAY è un array di elementi con lo stesso tipo di dati e un numero fisso di elementi (bounded).

```
--dichiarazione
TYPE type_name IS VARRAY(max_elements)
    OF element_type [NOT NULL];
--inizializzazione
varray_name type_name := type_name(element1,
element2, ...);
```

```
DECLARE
    TYPE t_nomi IS VARRAY(4)
        OF VARCHAR2(20) NOT NULL;
    a_nomi t_nomi  := t_nomi('Luca','Anna','Elia');
BEGIN
    DBMS_OUTPUT.PUT_LINE('A_nomi contiene ' || a_nomi.COUNT
|| ' elementi:');
    for i in a_nomi.FIRST..a_nomi.LAST -- 1..a_nomi.COUNT
    loop
        DBMS_OUTPUT.PUT_LINE('- ' || a_nomi(i));
    end loop;
END;
```

Array di record

- Esempio di VARRAY di record

```
DECLARE
TYPE r_article IS RECORD(
        num article.articlenum%TYPE,
        title article.title%TYPE
    ); -- definizione record

    TYPE t_article IS VARRAY(5) OF r_article; -- definizione array

    a_article t_article := t_article(); -- dichiarazione array
BEGIN
    a_article.EXTEND; -- usa EXTEND per aggiungere un'istanza
    a_article(a_article.LAST).num := 1; -- LAST = ultimo record
    a_article(a_article.LAST).title  := 'Titolo 1';
    …
    DBMS_OUTPUT.PUT_LINE('Numero articoli = ' || a_article.COUNT);
END;
```

Array di record

- Un array di record si può dichiarare a partire da una tabella

(ad esempio per caricare dati da un cursore)

```
DECLARE
    TYPE t_article IS VARRAY(5) OF article%ROWTYPE; -- def. array
    a_article t_article := t_article(); -- dichiarazione array
BEGIN
 FOR v_article IN (select * from article) LOOP - cursore implicito
    a_article.EXTEND;
    a_article(a_article.LAST).articlenum := v_article.articlenum;
    a_article(a_article.LAST).title  := v_article.title;
 END LOOP;
 DBMS_OUTPUT.PUT_LINE('Numero articoli = ' || a_article.COUNT);
END;
```

- Altri metodi per gestione di array:
  - DELETE: elimina tutti gli elementi `a_article.DELETE`
  - TRIM(n): rimuove n elementi dalla fine `a_article.TRIM`

Elenco dei metodi

Method	Description	SYNTAX
EXISTS (n)	This method will return Boolean results. It will return 'TRUE' if the n^{th} element exists in that collection, else it will return FALSE. Only EXISTS functions can be used in uninitialized collection	<collection_name>.EXISTS(element_position)
COUNT	Gives the total count of the elements present in a collection	<collection_name>.COUNT
LIMIT	It returns the maximum size of the collection. For Varray, it will return the fixed size that has been defined. For Nested table and Index-by-table, it gives NULL	<collection_name>.LIMIT
FIRST	Returns the value of the first index variable(subscript) of the collections	<collection_name>.FIRST
LAST	Returns the value of the last index variable(subscript) of the collections	<collection_name>.LAST
PRIOR (n)	Returns precedes index variable in a collection of the n^{th} element. If there is no precedes index value NULL is returned	<collection_name>.PRIOR(n)
NEXT (n)	Returns succeeds index variable in a collection of the n^{th} element. If there is no succeeds index value NULL is returned	<collection_name>.NEXT(n)
EXTEND	Extends one element in a collection at the end	<collection_name>.EXTEND
EXTEND (n)	Extends n elements at the end of a collection	<collection_name>.EXTEND(n)
EXTEND (n,i)	Extends n copies of the i^{th} element at the end of the collection	<collection_name>.EXTEND(n,i)
TRIM	Removes one element from the end of the collection	<collection_name>.TRIM
TRIM (n)	Removes n elements from the end of collection	<collection_name>.TRIM (n)
DELETE	Deletes all the elements from the collection. Makes the collection empty	<collection_name>.DELETE
DELETE (n)	Deletes the nth element from the collection. If the n^{th} element is NULL, then this will do nothing	<collection_name>.DELETE(n)
DELETE (m,n)	Deletes the element in the range m^{th} to n^{th} in the collection	<collection_name>.DELETE(m,n)

Procedurale vs Dichiarativo

- La modalità di calcolo da preferire è quella che massimizza le prestazioni (purché non complichi eccessivamente il codice)
- La principale regola di massima prevede che sia demandata all'ottimizzatore la modalità di accesso ai dati
  - E' meglio far eseguire al sistema una query complessa piuttosto che molte query semplici
  - Una valutazione più approfondita richiede di conoscere le modalità di accesso e di ottimizzazione utilizzate dal DBMS… E' anche per questo motivo che le studieremo

Procedurale vs Dichiarativo

- Un esempio: restituire in output separatamente l'importo di tutte le fatture con codice compreso tra 1 e 5

```
CURSOR cursore_importi IS
   SELECT D_NUMF,sum(D_QTA*D_PREZZO) as IMPORTO
   FROM   dettaglio
        WHERE D_NUMF BETWEEN 1 AND 5
        GROUP BY D_NUMF;
…
  LOOP
     FETCH cursore_importi into vr_importi;
     EXIT WHEN cursore_importi%NOTFOUND;
     DBMS_OUTPUT.PUT_LINE('La fattura: ' || vr_importi.D_NUMF
              || ' e'' di importo: ' || vr_importi.IMPORTO);
END LOOP;
```

Procedurale vs Dichiarativo

- Un esempio: restituire in output separatamente l'importo di tutte le fatture con codice compreso tra 1 e 5

```
FOR counter IN 1.. 5 LOOP
        SELECT sum(D_QTA*D_PREZZO) as IMPORTO INTO v_importo
        FROM   dettaglio
        WHERE D_NUMF = counter;
        DBMS_OUTPUT.PUT_LINE('La fattura: ' || counter
                || ' e'' di importo: ' || v_importo);
   END LOOP;
```

- **Meno efficiente scandisce il database 5 volte (in assenza di indici)**

Procedurale vs Dichiarativo

- Un esempio: restituire in output l'importo totale delle fatture che hanno singolarmente un importo > 1000 e <= 1000

```
CURSOR cursore_importi IS
       SELECT D_NUMF,sum(D_QTA*D_PREZZO) as IMPORTO
       FROM   dettaglio
       group by D_NUMF;
…
     open cursore_importi;
     LOOP
        FETCH cursore_importi into vr_importi;
        EXIT WHEN cursore_importi%NOTFOUND;
        if vr_importi.IMPORTO < 1000 then
          v_TotSmall := v_TotSmall + vr_importi.IMPORTO;
        else
          v_TotBig := v_TotBig + vr_importi.IMPORTO;
        end if;
    END LOOP;
```

Procedurale vs Dichiarativo

- Un esempio: restituire in output l'importo totale delle fatture che hanno singolarmente un importo > 1000 e <= 1000

```
SELECT SUM(IMPORTO) INTO v_TotBig
FROM (SELECT D_NUMF,sum(D_QTA*D_PREZZO) as IMPORTO
      FROM   dettaglio
      GROUP BY D_NUMF
      HAVING IMPORTO > 1000);
…
SELECT SUM(IMPORTO) INTO v_TotSmall
FROM (SELECT D_NUMF,sum(D_QTA*D_PREZZO) as IMPORTO
      FROM   dettaglio
      GROUP BY D_NUMF
      HAVING IMPORTO <= 1000);
```

- **Meno efficiente scandisce 2 volte il database: il calcolo della clausola HAVING non può sfruttare strutture a indice**

Sommario

- Tipi di cursore:
  - Impliciti: Utilizzati in tutti i comandi DML e per le query single-row.
  - Espliciti: Utilizzabili per le query a 0,1 o più righe.
- I cursori espliciti devono essere gestiti dal codice
  - DECLARE
  - OPEN
  - FETCH
  - CLOSE
- Lo stato del cursore può essere valutato utilizzando i suoi attributi

Esercizi

10. Scrivere una procedura che visualizza il nome del cliente associato a un ordine. Gestire il caso di ordine non presente.
11. Scrivere una procedura che stampa i fornitori e l'elenco dei prodotti forniti
12. Scrivere una procedura che restituisca separatamente il conteggio dei prodotti in tre fasce di prezzo date in input.
13. Scrivere una funzione che verifichi che un certo prodotto P sia presente in quantità > Q
14. Definire una tabella di appoggio:
    REORDER(idSupplier, idProduct, quantity, date)
    Scrivere una procedura che inserisce in REORDER un record per ciascun prodotto per cui la quantità è inferiore al livello di riordino