

PL/SQL - Oracle

PL - Procedural Language

SQL- Structured Query Language

PL/SQL – Referências

- ▣ Esta apostila apresenta conteúdos extraídos de materiais diversos:
 - ✓ Prof. Edmundo Spoto – UNIVASF – CECOMP
 - ✓ Prof. Oscar F. dos Santos – UNINOVE
 - ✓ Prof. Marcos Alexandruk – UNINOVE
 - ✓ Oracle.ppt - <http://dc265.4shared.com/doc/uBmzoijsco/preview.html> - PUC - RS
 - ✓ <http://phpdba.wordpress.com/2011/04/03/plsql-bloco-anonimo/>

PL/SQL - Conceito

- A **PL/SQL** é uma linguagem de programação sofisticada utilizada para acessar um banco de dados Oracle a partir de vários ambientes.
- Ela é integrada com o servidor do banco de dados de modo que o código PL/SQL possa ser processado de maneira rápida e eficiente.
- Pode-se usar comandos SQL para manipular os dados da base ORACLE e fluxos de controle para processar os dados.
- Pode-se declarar constantes e variáveis, definir subprogramas (procedures ou funções) e controlar erros de execução.

PL/SQL - Conceito

- Um código PL/SQL pode ser armazenado como um objeto dentro do banco de dados. Seriam as famosas Procedures, Functions e Triggers. Esses objetos podem ainda ser agrupados em Packages (pacotes), para facilitar a lógica e o gerenciamento de um grupo de blocos que possuem relacionamento entre si ou compartilham fatores em comum.
- Eventualmente, podemos sentir a necessidade de executar um código PL/SQL apenas uma vez, para algum teste ou para uma correção de um dado, por exemplo. Nesses casos (e em outros que julgar necessário) você pode usar um bloco anônimo. É um bloco PL/SQL que não será armazenado definitivamente no banco. O bloco será interpretado, executado e depois será descartado.

PL/SQL - Performance

□ A **PL/SQL** pode:

- ✓ **Reduzir o tráfego** na rede pelo envio de um bloco contendo diversos comandos de SQL agrupados em blocos para o Oracle;
- ✓ **Reduzir** não só o tráfego na rede como também a programação quando estabelecemos ações que podem ser compartilhadas em diversas aplicações (uso de stored procedure).

PL/SQL – Estrutura do Bloco Anônimo

- A PL/SQL é uma estrutura em blocos, ou seja, as unidades básicas (procedures, funções, etc.) podem conter qualquer número de sub-blocos.
- Cada bloco é composto de três partes:

DECLARE → opcional

<definição dos objetos PL/SQL que serão utilizados neste bloco>

BEGIN → obrigatório

<lógica, ações executáveis>

EXCEPTION → opcional

< o que fazer se uma ação executada causar um erro >

END; → termina a definição do bloco

PL/SQL – Identificadores

- Um identificador consiste em uma letra seguida de outras letras, números, \$ (dólar), _ (sublinhado) e # (símbolo numérico). Possui um limite máximo de 30 caracteres.
 - As letras podem ser maiúsculas ou minúsculas indiscriminadamente.
 - Não utilize palavras reservadas na declaração de variáveis como IF, END, DO, WHILE, ELSE, etc...., mas caso seja necessário, coloque a palavra entre aspas.
Ex: "IF"
 - Evite usar o mesmo nome de colunas das tabelas
- Dica:** use **v_nome** para diferenciar o nome da coluna da tabela com o nome da variável

PL/SQL – Comentários

- Um comentário em PL/SQL pode ser informado de duas maneiras:
 - ✓ Dois hífen em qualquer ponto da linha torna o restante da linha como comentário (--)
 - ✓ **/*** (início) e ***/** (fim) marcam um bloco de comentários (quando o comentário utiliza mais de uma linha ou o comentário está no meio de um comando)

Exemplo: SQL> DECLARE
2 v_valor VARCHAR(10);
3 v_end VARCHAR(20);
4 BEGIN
5 v_valor := -- A atribuição será feita na outra linha
6 'Rua A';
7 v_end := /* A atribuição virá a seguir */ 'Rua B';
8 END;

Obs: A indicação de fim de linha de comando em PL/SQL é feita com um ponto-e-vírgula (;)

PL/SQL – Interagindo com o usuário

- A PL/SQL não tem nenhuma funcionalidade de entrada ou de saída construída diretamente na linguagem. Para retificar isso, o SQL*Plus, em combinação com o pacote DBMS_OUTPUT, fornece a capacidade de dar saída para mensagens em tela. Isso é feito em dois passos:

- ✓ Permitir a saída no SQL*Plus com o comando *set serveroutput on*:

SET SERVEROUTPUT {ON | OFF} [SIZE n]

onde **n** é o tamanho do buffer de saída. Seu valor padrão é 2.000 bytes

- ✓ Dentro de um bloco PL/SQL utilize a procedure PUT para mensagem de saída :

DBMS_OUTPUT.PUT_LINE ('mensagem')

PL/SQL – Variáveis de Memória

Variáveis podem ser utilizadas para:

- Armazenar dados temporários
- Manipulação de valores armazenados
- Reusabilidade (não é necessário consultar repetidas vezes os dados armazenados)
- Facilidade de manutenção

PL/SQL – Variáveis de Memória

- ❑ Variáveis PL/SQL suportam todos os tipos de dados da linguagem SQL e novos tipos definidos para o PL/SQL, como BOOLEAN (admite os valores TRUE, FALSE ou NULL).
- ❑ As variáveis **podem receber ou não** valor inicial.
- ❑ Caso a declaração da variável receba a cláusula **Not Null**, a atribuição de valor inicial se torna obrigatória.
- ❑ Quando **não informamos valor inicial** para uma variável, seu valor é considerado desconhecido, ou seja, **Null**.
- ❑ Uma variável declarada em um bloco **é visível nesse e em todos os blocos subordinados a ele**.
- ❑ Uma variável declarada em um bloco **deixa de existir** quando o bloco termina.

PL/SQL – Variáveis de Memória - Atribuição

- A atribuição de valor a uma variável é feita com a notação **:=**.
- As variáveis e constantes são criadas e recebem seu valor inicial cada vez que for iniciado o bloco no qual estão declaradas.
- A PL/SQL **não declara, implicitamente**, variáveis como ocorre em outras linguagens. É muito importante, portanto, que as variáveis sejam inicializadas antes do uso.
- Se um identificador de uma expressão de um comando SQL não é um coluna, então o ORACLE assume que ele é uma variável (ou constante) PL/SQL.

PL/SQL – Variáveis de Memória

- As variáveis devem ser declaradas na seção **DECLARE**.

✓ Exemplo:

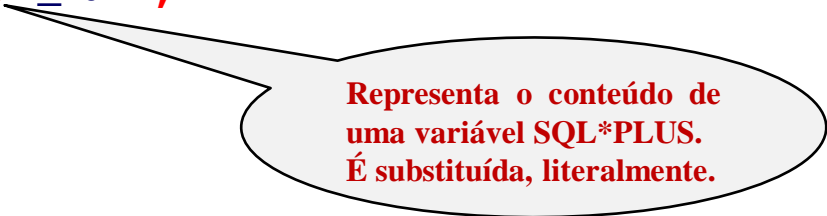
```
DECLARE
    v_var1      number(6);
    v_var2      varchar(20)  := 'Teste';
    v_var3      date         NOT NULL := sysdate;
BEGIN
    :
    :
END ;
```

PL/SQL – Variáveis de Memória

- As variáveis devem ser declaradas na seção **DECLARE**.

✓ Exemplo:

```
DECLARE
  v_var1      number(6);
  v_var2      varchar(20)  := 'Teste';
  v_var3      date         NOT NULL := sysdate;
  v_var4      number(5,2)  := &v_var4 ;
BEGIN
  :
  :
END ;
```



Representa o conteúdo de
uma variável SQL*PLUS.
É substituída, literalmente.

PL/SQL – Constantes

- **CONSTANTE** é uma variável que não pode ter seu valor alterado.

✓ Exemplo:

```
DECLARE
    PI    CONSTANT number(7,6)  :=  3.141592;
    :
    :
BEGIN
    :
    :
END ;
```

- *Constantes* devem ter seus valores atribuídos na declaração, **necessariamente**.

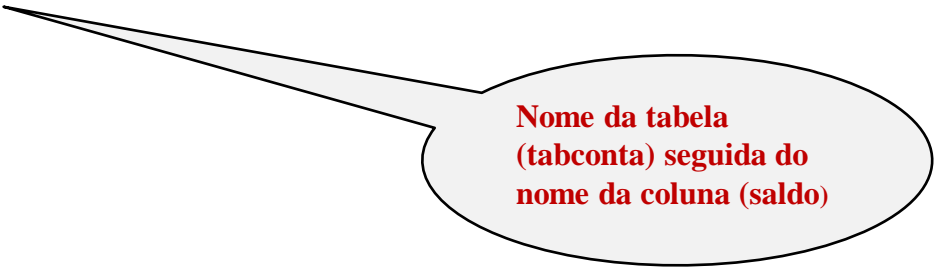
PL/SQL – Herança de Tipos

□ É possível declarar uma variável com o mesmo tipo de:

- ✓ Uma outra variável pré-declarada.
- ✓ Uma coluna existente em uma tabela.

✓ Exemplo:

```
DECLARE
    v_salario      number(9,2);
    v_comissao     v_salario%TYPE;
    v_valsaldo     tabconta.saldo%TYPE;
BEGIN
    :
    :
END ;
```



Nome da tabela
(tabconta) seguida do
nome da coluna (saldo)

PL/SQL – Herança de Tipos

- ❑ O atributo **%TYPE** copia o tipo de dado de uma variável ou coluna de uma tabela do banco de dados, podendo, dessa forma, declarar variáveis baseadas no tipo de outras variáveis ou de campos de tabelas.
- ❑ Isso é bastante utilizado quando uma variável recebe valor vindo de campo de uma tabela.
- ❑ A utilização de **%TYPE** favorece a independência de dados, uma vez que **não precisamos saber exatamente o tipo de dado da variável a ser declarada** e, caso haja modificações no objeto original, a modificação da variável CÓPIA é feita automaticamente.

PL/SQL – Comandos DML

- ❑ PL/SQL foi criado para flexibilizar a manipulação de dados através de comandos SQL.
- ❑ Só é possível utilizar os comandos SQL de manipulação de dados (SELECT, INSERT, UPDATE e DELETE) dentro de PL/SQL, além dos comandos de controle de transações (ROLLBACK, COMMIT e SAVEPOINT).
- ❑ Tudo o que se pode utilizar em cláusulas WHERE na linguagem SQL pode-se utilizar também em PL/SQL.
- ❑ É possível utilizar todas as funções numéricas, de caracteres, datas e conversões de dados disponíveis na linguagem SQL.
- ❑ É possível utilizar as funções de grupo da linguagem SQL.

PL/SQL – Comandos DML

- ✓ **Exemplo:** Criar a tabela ALUNO, inserir registros e pesquisá-los. (Para este exemplo, os comandos CREATE TABLE e INSERT abaixo devem ser executados fora do bloco PL/SQL).

```
CREATE TABLE ALUNO ( RA NUMBER(5), NOME VARCHAR2(40));
```

```
INSERT INTO ALUNO VALUES (1,'ANTONIO');
```

```
INSERT INTO ALUNO VALUES (2,'BEATRIZ');
```

DECLARE

```
V_RA      ALUNO.RA%TYPE;
```

```
V_NOME    ALUNO.NOME%TYPE;
```

BEGIN

```
SELECT RA, NOME INTO V_RA, V_NOME FROM ALUNO WHERE RA=1;
```

```
DBMS_OUTPUT.PUT_LINE(V_RA || ' - ' || V_NOME);
```

```
END;
```

```
/
```

Obs: Este tipo de construção só funciona quando o SELECT devolva uma única linha. No caso de várias linhas, será usado um CURSOR, explicado posteriormente.

PL/SQL – Comandos DML - SELECT

- ❑ O comando SELECT dentro da PL/SQL deverá receber obrigatoriamente a cláusula **INTO** para que o resultado seja armazenado em variáveis e dessa forma, poder manusear os dados obtidos.
- ❑ Devemos informar *uma área de recepção* capaz de comportar todos os dados lidos; caso contrário, receberemos erro na execução.
- ❑ A cláusula **Into** *segue imediatamente* a lista de campos da cláusula **Select** e *precede* a cláusula **From**.
- ❑ O comando SELECT deverá retornar apenas uma linha. Caso mais de uma linha seja retornada apresentará o erro **too_many_rows** e se não retornar nenhuma linha apresentará o erro **no_data_found**.

PL/SQL – Comandos DML - SELECT

✓ Exemplo:

```
SELECT RA, NOME INTO V_RA, V_NOME FROM ALUNO  
WHERE RA=1;
```

- Caso a cláusula WHERE fosse alterada para: **WHERE** RA=1 **OR** RA=2 o comando *SELECT* retornaria mais de uma linha, exibindo a mensagem:
ORA-01422: a extração exata retorna mais do que o número solicitado de linhas
- Caso a cláusula WHERE fosse alterada para: **WHERE** RA=3 o comando *SELECT* não encontraria linha para retornar, exibindo a mensagem:
ORA-01403: dados não encontrados
- Caso a cláusula WHERE fosse alterada para: **WHERE** RA=&V_RA_Aluno e a variável V_RA_Aluno tivesse sido declarada, o programa pediria para que o usuário informasse o valor do RA, entendendo o uso do & como parâmetro a ser informado em tempo de execução.

PL/SQL – Comandos DML

- ✓ Exemplo: Alterar o nome do aluno cujo RA=2 para 'ERNESTO' na tabela ALUNO

BEGIN

UPDATE ALUNO **SET** NOME='ERNESTO' **WHERE** RA=2 ;

END;

/

- ✓ Exemplo: Excluir da tabela ALUNO o registro cujo RA=2

BEGIN

DELETE FROM ALUNO **WHERE** RA=2 ;

END;

/

PL/SQL – Tratamento de Exceções

- Durante a execução de um bloco PL/SQL, se acontecer alguma “situação anormal” (por exemplo, uma divisão por zero), o fluxo de execução é transferido diretamente para o tratador de exceções daquele bloco (como se fosse um GOTO).
- Um tratador de exceções é definido da seguinte forma:

BEGIN

:

:

EXCEPTION

WHEN *nome_exceção1* **THEN**

COMANDO1_1

COMANDO1_2

:

WHEN *nome_exceção2* **THEN**

COMANDO1_1

COMANDO1_2

:

END;

PL/SQL – Exceções pré-definidas

- ▣ Existe um conjunto de situações anormais, ou exceções pré-definidas:
 - ✓ **DUP_VAL_ON_INDEX** - Quando ocorrer violação de um **índice único** através de um INSERT ou UPDATE.
 - ✓ **INVALID_CURSOR** – Quando um dos atributos FOUND, NOTFOUND ou ROWCOUNT é consultado para um cursor fechado.
 - ✓ **INVALID_NUMBER** – Quando a conversão de uma cadeia de caracteres para um número dá errado.
 - ✓ **NO_DATA_FOUND** – Um comando SELECT não retornou nenhuma linha.
 - ✓ **TOO_MANY_ROWS** – Um comando SELECT retornou mais de uma linha.
 - ✓ **VALUE_ERROR** – Quando acontece um erro de “OVERFLOW” ou uma cadeia de caracteres é “TRUNCADA”.
 - ✓ **ZERO_DIVIDE** – Quando há uma tentativa de divisão por zero.
 - ✓ **OTHERS** – Qualquer outra exceção não especificada anteriormente. Deve ser a última cláusula WHEN especificada.

PL/SQL – Exceções pré-definidas

✓ Exemplo:

DECLARE

V_RA ALUNO.RA%TYPE;

V_NOME ALUNO.NOME%TYPE;

BEGIN

SELECT RA, NOME INTO V_RA, V_NOME FROM ALUNO WHERE RA=30;

DBMS_OUTPUT.PUT_LINE(V_RA || ' - ' || V_NOME);

EXCEPTION

WHEN NO_DATA_FOUND THEN

DBMS_OUTPUT.PUT_LINE ('Não há nenhum aluno com este RA');

WHEN TOO_MANY_ROWS THEN

DBMS_OUTPUT.PUT_LINE ('Há mais de um aluno com este RA');

WHEN OTHERS THEN

DBMS_OUTPUT.PUT_LINE ('Erro desconhecido');

END;

PL/SQL – Comandos de Decisão - IF

- ❑ O comando **IF** verifica uma condição e dependendo do resultado realiza uma ou outra ação. Permite a execução condicional de uma determinada ação.
- ❑ Existem três sintaxes válidas de IF em PL/SQL.
 - ✓ **IF** *CONDIÇÃO* **THEN** *[CORPO DE COMANDOS]* **END IF;**
 - ✓ **IF** *CONDIÇÃO* **THEN** *[CORPO DE COMANDOS]*
ELSE *[CORPO DE COMANDOS DO ELSE]* **END IF;**
 - ✓ **IF** *CONDIÇÃO* **THEN** *[CORPO DE COMANDOS]*
ELSIF *CONDIÇÃO* **THEN** *[CORPO DE COMANDOS]* **END IF;**

onde:

- *Condição* – é uma expressão booleana que, quando nula (NULL) é tratada como FALSE.
- *Comando* – qualquer comando PL/SQL (inclusive IF).

PL/SQL – Comandos de Decisão - IF

✓ Exemplo 1:

DECLARE

```
var1 number(4) := &var1;  
var2 number(4);
```

BEGIN

```
IF var1 > 10 THEN  
    var2 := var1 + 20;
```

```
END IF;
```

```
dbms_output.put_line('O valor de var2 é: ' || var2);
```

```
END;
```

✓ Exemplo 2:

DECLARE

```
var1 number(4) := &var1;  
var2 number(4);
```

BEGIN

```
IF var1 > 10 THEN  
    var2 := var1 + 20;
```

```
ELSE
```

```
    var2 := var1 * var1;
```

```
END IF;
```

```
dbms_output.put_line('O valor de var2 é: ' || var2);
```

```
END;
```

PL/SQL – Comandos de Decisão - IF

✓ Estrutura:

```
IF condição THEN  
    comandos;  
ELSIF condição THEN  
    comandos;  
ELSE  
    comandos;  
END IF;
```

PL/SQL – Comandos de Decisão - IF

✓ Exemplo 3:

DECLARE

var1 number(4) := &var1;

var2 number(4);

BEGIN

IF var1 > 10 **THEN**

var2 := var1 + 20;

ELSIF var1 between 7 and 9 **THEN**

var2 := var1 * 2;

ELSE

var2 := var1 * var1;

END IF;

dbms_output.put_line('O valor de var2 é: ' || var2);

END;

PL/SQL – Comandos de Decisão - IF

✓ Exemplo:

DECLARE

V_VALOR NUMBER(7,2) := &V_VALOR;

BEGIN

IF V_VALOR > 0 **THEN**

DBMS_OUTPUT.PUT_LINE(VALOR || ` Este valor é maior que zero`);

ELSIF V_VALOR = 0 **THEN**

DBMS_OUTPUT.PUT_LINE(VALOR || ` Este valor é igual a zero`);

ELSE

DBMS_OUTPUT.PUT_LINE(VALOR || ` Este valor é menor que zero`);

END IF;

END;

PL/SQL – Comandos de Decisão - IF

✓ Exemplo:

Executar os comandos CREATE TABLE e INSERT fora do bloco PL/SQL.

```
CREATE TABLE ALUNO ( RA NUMBER(9), NOTA NUMBER(3,1));
```

```
INSERT INTO ALUNO VALUES (1,4);
```

DECLARE

```
V_RA ALUNO.RA%TYPE := 1;
```

```
V_NOTA ALUNO.NOTA%TYPE;
```

```
V_CONCEITO VARCHAR2(12);
```

BEGIN

```
SELECT NOTA INTO V_NOTA FROM ALUNO WHERE RA = V_RA;
```

```
IF V_NOTA <= 5 THEN
```

```
    V_CONCEITO := 'REGULAR';
```

```
ELSIF V_NOTA < 7 THEN
```

```
    V_CONCEITO := 'BOM';
```

```
ELSE V_CONCEITO := 'EXCELENTE';
```

```
END IF;
```

```
DBMS_OUTPUT.PUT_LINE ('Conceito: ' || V_CONCEITO);
```

```
END;
```

PL/SQL – Comandos de Decisão - CASE

- Retorna determinado resultado de acordo com o valor da variável de comparação.

✓ Estrutura:

```
VARIÁVEL :=  
CASE  
    WHEN expressão_1 THEN declaração_1  
    WHEN expressão_2 THEN declaração_2  
    . . . .  
ELSE  
    declaração_n  
END;
```


PL/SQL – Comandos de Decisão - CASE

✓ Exemplo 1:

DECLARE

var1 number(4) := &var1;

var2 number(4);

BEGIN

var2 :=

CASE

WHEN var1 > 10 **THEN** var1 + 20

WHEN var1 between 7 and 9 **THEN** var1 * 2

ELSE

var1 * var1

END;

dbms_output.put_line('O valor de var2 é: ' || var2);

END;

PL/SQL – Comandos de Decisão - CASE

✓ Exemplo 2:

DECLARE

```
v_ra ALUNO.RA%TYPE := 1;  
v_nota ALUNO.NOTA%TYPE;  
v_conceito VARCHAR(12);
```

BEGIN

```
SELECT nota INTO v_nota FROM ALUNO WHERE RA = v_ra;
```

```
v_conceito :=
```

CASE

```
WHEN v_nota <= 5 THEN 'REGULAR'
```

```
WHEN v_nota < 7 THEN 'BOM'
```

ELSE

```
'EXCELENTE'
```

END;

```
dbms_output.put_line('Conceito: ' || v_conceito);
```

END;

PL/SQL – Comandos de Decisão - CASE

- Caso não seja colocada a cláusula *ELSE* e nenhuma condição da estrutura CASE for validada, a PL/SQL interromperá a execução com o erro CASE_NOT_FOUND ou ORA-6592.

✓ Exemplo:

DECLARE

v_TestVar NUMBER := 1;

BEGIN

CASE v_TestVar

WHEN 2 THEN DBMS_OUTPUT.PUT_LINE('Two!');

WHEN 3 THEN DBMS_OUTPUT.PUT_LINE('Three!');

WHEN 4 THEN DBMS_OUTPUT.PUT_LINE('Four!');

END CASE;

END;

/

*

ERRO na linha 1:

ORA-06592: CASE não encontrado ao executar a instrução CASE

Obs: Usar EXCEPTION para tratar o erro substitui o uso da cláusula ELSE (veja Tratamento de Exceções na próxima página).

PL/SQL – Comandos de Decisão - CASE

- Existem duas maneiras de escrever um bloco PL/SQL com estrutura CASE.
 - CASE** finalizado com **END** - usado quando a estrutura vai *retornar um valor* que será atribuído numa variável. **Não deve** ser colocado (;) dentro da estrutura.

✓ Exemplo:

DECLARE

```
v_ra ALUNO.RA%TYPE := 1;  
v_nota ALUNO.NOTA%TYPE;  
v_conceito VARCHAR(12);
```

BEGIN

```
SELECT nota INTO v_nota FROM ALUNO WHERE RA = v_ra;
```

```
v_conceito :=
```

CASE

```
    WHEN v_nota <= 5 THEN 'REGULAR'
```

```
    WHEN v_nota < 7  THEN 'BOM'
```

```
    ELSE
```

```
        'EXCELENTE'
```

END;

```
dbms_output.put_line('Conceito: ' || v_conceito);
```

END;

PL/SQL – Comandos de Decisão - CASE

- **CASE** finalizado com **END CASE** - usado quando a *estrutura* é apenas *de controle* dentro de um algoritmo para que *instruções sejam executadas*, sem retornar valor. **Deve ser** colocado (;) dentro da estrutura.

✓ **Exemplo:**

DECLARE

v_TestVar NUMBER := 1;

BEGIN

CASE v_TestVar

WHEN 2 THEN DBMS_OUTPUT.PUT_LINE('Two!');

WHEN 3 THEN DBMS_OUTPUT.PUT_LINE('Three!');

WHEN 4 THEN DBMS_OUTPUT.PUT_LINE('Four!');

END CASE;

END;

Obs: Este tipo de estrutura é similar a estrutura de decisão **IF**, onde todas as instruções dentro do THEN, ELSE e ELSIF tem ponto-e-vírgula (;).

PL/SQL – Exceções definidas pelo usuário

- Além dos erros tratados automaticamente pelo Oracle, regras de negócio específicas podem ser tratadas.
- As exceções definidas pelo usuário devem ser declaradas e chamadas explicitamente pelo comando **RAISE**.

✓ Exemplo:

DECLARE

nome_excecao exception;

BEGIN

....

If then

Raise nome_excecao;

End IF;

....

EXCEPTION

When nome_excecao **then**

.....

END;

PL/SQL – Exceções definidas pelo usuário

✓ Exemplo:

Declare

```
V_conta number(2);  
Conta_aluno exception;
```

Begin

```
Select count (Ra) into v_conta from aluno;  
If v_conta = 10 then  
    Raise conta_aluno;  
Else  
    Insert into aluno values (20, 'silva');  
End if;
```

Exception

```
When conta_aluno then
```

```
    Dbms_output.put_line ('Não foi possível incluir, turma cheia');
```

```
End;
```

PL/SQL – Comandos de Repetição

- Existem três estruturas de repetição num bloco PL/SQL:
 - ✓ LOOP
 - ✓ WHILE
 - ✓ FOR

PL/SQL – Comandos de Repetição - LOOP

- Executa uma relação de comandos até que uma instrução de saída (**EXIT**) seja encontrada.

✓ Estutura:

LOOP

sequência_de_instruções

IF condição_de_saída **THEN**

EXIT;

END IF;

END LOOP;

PL/SQL – Comandos de Repetição - LOOP

✓ Exemplo 1:

DECLARE

V_AUX NUMBER(2) := 0;

BEGIN

LOOP

V_AUX := V_AUX + 1;

DBMS_OUTPUT.PUT_LINE (V_AUX);

IF V_AUX = 10 THEN

EXIT;

END IF;

END LOOP;

END;

PL/SQL – Comandos de Repetição - WHILE

- Repete um bloco de comandos enquanto a *condição* que segue o comando WHILE for verdadeira.

✓ Estrutura:

```
WHILE condição LOOP  
    sequência_de_instruções;  
END LOOP;
```

Obs: Se a condição de loop não for avaliada como **TRUE** na primeira vez que ela for verificada, o loop não será executado.

PL/SQL – Comandos de Repetição - WHILE

✓ Exemplo 1:

DECLARE

V_AUX NUMBER(2) := 0;

BEGIN

WHILE V_AUX < 10 **LOOP**

V_AUX := V_AUX + 1;

DBMS_OUTPUT.PUT_LINE (V_AUX);

END LOOP;

END;

PL/SQL – Comandos de Repetição - WHILE

✓ Exemplo 2:

DECLARE

V_RA_FINAL ALUNO.RA%TYPE := 1;

V_AUX V_RA_FINAL%TYPE := 0;

BEGIN

SELECT COUNT(RA) INTO V_RA_FINAL FROM ALUNO;

WHILE V_AUX < V_RA_FINAL **LOOP**

V_AUX := V_AUX + 1;

DBMS_OUTPUT.PUT_LINE ('Total de alunos: ' || V_AUX);

END LOOP;

END;

PL/SQL – Comandos de Repetição - FOR

- Repete um bloco de comandos n vezes, ou seja, até que a variável contadora atinja o seu valor final.
- A variável contadora não deve ser declarada na seção DECLARE e deixará de existir após a execução do comando END LOOP.

✓ Estutura:

```
FOR v_contador IN valor_inicial..valor_final LOOP  
    sequência_de_instruções  
END LOOP;
```

PL/SQL – Comandos de Repetição - FOR

✓ Exemplo 1:

DECLARE

V_AUX NUMBER(2) := 0;

BEGIN

-- não é necessário declarar a variável v_contador

-- ela será automaticamente declarada como BINARY_INTEGER

FOR V_CONTADOR IN 1..10 LOOP

V_AUX := V_AUX +1;

DBMS_OUTPUT.PUT_LINE (V_AUX);

END LOOP;

END;

PL/SQL – Comandos de Repetição - FOR

✓ Exemplo 2:

DECLARE

V_RA_INICIAL ALUNO.RA%TYPE := 1;

V_RA_FINAL V_RA_INICIAL%TYPE;

V_AUX V_RA_INICIAL%TYPE := 0;

BEGIN

SELECT COUNT(RA) INTO V_RA_FINAL FROM ALUNO;

FOR V_CONTADOR **IN** V_RA_INICIAL..V_RA_FINAL **LOOP**

V_AUX := V_AUX + 1;

DBMS_OUTPUT.PUT_LINE ('Total de alunos: ' || V_AUX);

END LOOP;

END;

PL/SQL – Comandos de Repetição - FOR

- Se a palavra **REVERSE** estiver presente no loop FOR, o índice de loop irá refazer a partir do valor alto para o valor baixo.

✓ Exemplo:

DECLARE

v_Baixo NUMBER := 10;

v_Alto NUMBER := 40;

BEGIN

FOR v_cont **IN REVERSE** v_Baixo .. v_Alto **LOOP**

INSERT INTO tab_exemplo VALUES(v_cont, 'Teste');

DBMS_OUTPUT.PUT_LINE ('V_cont = ' || v_cont || ' - Registro inserido com sucesso');

END LOOP;

END;

Obs: Antes de executar o bloco PL/SQL acima, crie a tabela "tab_exemplo", conforme comando abaixo:

CREATE TABLE tab_exemplo (Cod NUMBER(2), Descricao VARCHAR(10));

PL/SQL – Comandos de Repetição

- Se desejado, as instruções EXIT ou EXIT WHEN podem ainda ser utilizadas dentro de qualquer estrutura de repetição (LOOP, FOR, WHILE) para sair prematuramente do loop.

✓ Exemplo 1: Utilizando *EXIT*

DECLARE

V_AUX NUMBER(2) := 0;

BEGIN

FOR V_CONTADOR **IN** 1..15 **LOOP**

V_AUX := V_AUX + 1;

DBMS_OUTPUT.PUT_LINE (V_AUX);

IF V_CONTADOR = 10 THEN

EXIT;

END IF;

END LOOP;

END;

PL/SQL – Comandos de Repetição

- ✓ Exemplo 2: Utilizando *EXIT WHEN*

DECLARE

V_AUX NUMBER(2) := 0;

BEGIN

FOR V_CONTADOR IN 1..15 **LOOP**

V_AUX := V_AUX + 1;

DBMS_OUTPUT.PUT_LINE (V_AUX);

EXIT WHEN V_CONTADOR = 10;

END LOOP;

END;

PL/SQL – Labels

- Utilizados para nomear blocos ou sub-blocos.
- Devem localizar-se antes do início do bloco e preceder pelo menos um comando.
- Se não houver necessidade de nenhum comando após o label, deve-se utilizar o comando **NULL**.

✓ Exemplo:

```
<<NOME_DO_LABEL>>  
DECLARE  
  
...  
BEGIN  
    relação_de_comandos  
    <<NOME_DO_LABEL>>  
    relação_de_comandos  
END;
```

PL/SQL – Labels

- Um label também pode ser aplicado a um comando de repetição LOOP.

✓ Exemplo:

```
<<PRINCIPAL>>  
LOOP  
    ...  
    LOOP  
        ...  
        -- SAIR DOS DOIS LOOPS  
        EXIT PRINCIPAL WHEN ...  
    END LOOP;  
END LOOP PRINCIPAL;
```

PL/SQL – GOTO

- Utilizado para desviar um fluxo de um bloco PL/SQL para determinado label.

GOTO nome_do_label

Nota: Um Label deve ser codificado entre <<L>>. No comando **GOTO**, os símbolos <<L>> não devem ser usados

PL/SQL – GOTO

✓ Exemplo 1:

<<PRINCIPAL>>

DECLARE

V_NOME ALUNO.NOME%TYPE;

V_CONTA NUMBER(2);

BEGIN

SELECT COUNT(RA) INTO V_CONTA FROM ALUNO;

IF V_CONTA = 10 THEN

GOTO FIM;

ELSE

INSERT INTO ALUNO VALUES (20,'SILVA');

END IF;

<<FIM>>

DBMS_OUTPUT.PUT_LINE('Fim do programa');

END;

PL/SQL – GOTO

✓ Exemplo 2:

<<PRINCIPAL>>

DECLARE

V_NOME ALUNO.NOME%TYPE;

BEGIN

SELECT NOME INTO V_NOME FROM ALUNO WHERE NOME LIKE '&NOME_ALUNO';

FOR V_CONTADOR IN 1..5 **LOOP**

<<SECUNDARIO>>

DECLARE

V_NOME VARCHAR2(40);

BEGIN

SELECT NOME INTO V_NOME FROM ALUNO WHERE RA=V_CONTADOR;

IF V_NOME = PRINCIPAL.V_NOME THEN

DBMS_OUTPUT.PUT_LINE('Está entre os 5 primeiros');

GOTO FIM;

END IF;

END;

END LOOP;

<<FIM>>

DBMS_OUTPUT.PUT_LINE('Fim do programa');

END;

PL/SQL – GOTO

- Existem algumas situações em que o comando **Goto** *não pode* ser usado: (Veja exemplo na próxima página)
 - ✓ Desviar o fluxo para dentro de um IF ou LOOP;
 - ✓ Desviar o fluxo de um IF para outro; (**GoTo Proximo_IF**)
 - ✓ Desviar o fluxo para dentro de um sub-bloco; (**GoTo subbloco**)
 - ✓ Desviar o fluxo para um bloco externo ao bloco corrente; (**GoTo Para_Fora**)
 - ✓ Desviar o fluxo de uma EXCEPTION para o bloco corrente e vice-versa. (**GoTo Inicio**)

PL/SQL – GOTO

✓ Exemplo:

```
SQL> DECLARE
2     PROCEDURE DESVIO IS
3     BEGIN
4         GOTO PARA_FORA;
5     END;
6     BEGIN
7         <<INICIO>>
8         IF TRUE THEN
9             GOTO PROXIMO_IF;
10        END IF;
11        IF FALSE THEN
12            <<PROXIMO_IF>>
13            GOTO SUBBLOCO;
14        END IF;
15        <<PARA_FORA>>
16        BEGIN
17            <<SUBBLOCO>>
18            NULL;
19        END;
20    EXCEPTION
21        WHEN OTHERS THEN
22            GOTO INICIO;
23    END;
```

GOTO PROXIMO_IF;

ERRO na linha 9:

ORA-06550: linha 9, coluna 7:

PLS-00375: instrução GOTO inválida;

este GOTO não pode ser ramificado para o label 'PROXIMO_IF'

ORA-06550: linha 11, coluna 13:

PL/SQL: Statement ignored

ORA-06550: linha 4, coluna 5:

PLS-00375: instrução GOTO inválida;

este GOTO não pode ser ramificado para o label 'PARA_FORA'

ORA-06550: linha 6, coluna 1:

PL/SQL: Statement ignored

ORA-06550: linha 22, coluna 5:

PLS-00375: instrução GOTO inválida;

este GOTO não pode ser ramificado para o label 'INICIO'

ORA-06550: linha 22, coluna 5:

PL/SQL: Statement ignored

PL/SQL – GOTO

- O comando **NULL**, explicitamente, indica que não há ação a ser feita. Serve para compor certas situações em que um comando é exigido, mas nenhuma ação é, realmente, necessária.

✓ **Exemplo:**

DECLARE

VEZ NUMBER := 1;

BEGIN

<< INICIO >>

VEZ := VEZ + 1;

IF VEZ > 10 THEN

GOTO FIM;

ELSE

VEZ := VEZ**2 - 1;

END IF;

GOTO INICIO;

<<FIM>>

NULL;

END;

Já vimos anteriormente que todo **Label** deve preceder um comando ou um bloco de PL/SQL.

No exemplo ao lado, a presença do comando **Null** teve a finalidade de validar o posicionamento do **Label Fim**. Caso o comando não fosse acionado, não poderíamos programar o **Fim** antes de **End** (não é um comando), que é, na verdade, fim de bloco.

PL/SQL – Declaração de Registros

- Também é possível, em PL/SQL, declarar registros (**%ROWTYPE**) com a mesma estrutura de uma tabela, visão ou conjunto de colunas retornadas por um cursor.
- Não é necessário conhecer a estrutura ou tipos de dados.
- A estrutura e os tipos de dados podem ser alterados.
- **%ROWTYPE** é útil para recuperar um registro em um comando SELECT.
- Como com %TYPE, qualquer restrição NOT NULL definida na coluna não é copiada para o registro.

PL/SQL – Declaração de Registros

✓ Exemplo:

DECLARE

Reg_conta Conta%ROWTYPE;

:
:
:

BEGIN

:

Reg_conta.saldo.....

:

END;

Nome da tabela

Representa o campo saldo
do registro Reg_conta, não
da tabela Conta.

- O registro criado tem campos com o mesmo nome e tipo das colunas da tabela ou visão.
- Registros também são utilizados para a manipulação de cursores, que serão vistos na sequência.
- A variável **Reg_conta** herdou o tipo correspondente a uma linha inteira da tabela **Conta**.

PL/SQL – Cursor Implícito e Explícito

- A fim de processar uma instrução SQL, o Oracle alocará uma área de memória conhecida como área de contexto. A área de contexto contém as informações necessárias para completar o processamento, incluindo o número de linhas processadas pela instrução e, no caso de uma consulta, o conjunto ativo, que é o conjunto de linhas retornadas pela consulta.
- Portanto, cursores são áreas compostas de linhas e colunas em memória que servem para armazenar o resultado de uma expressão que retorna 0(zero) ou mais linhas.
- Cursores são trechos alocados de memória destinados a processar as declarações SELECT. Podem ser definidos pelo próprio PL/SQL, chamados de **Cursores Implícitos**, ou podem ser definidos manualmente, são os chamados de **Cursores Explícitos**.

PL/SQL – Cursor Implícito

- “**SQL**” é o identificador de “**cursor implícito**”. Todo **comando DML**, automaticamente (implicitamente) abre um cursor interno no SGBD sempre identificado como “**SQL**”.
 - ▣ É útil para testar o que aconteceu com o último comando DML executado, através das “propriedades” do cursor, chamadas de “atributos do cursor”. Todos eles têm seu nome começando com o símbolo “**%**” e são referenciados colocando-se o **nome do cursor** antes do “**%**”.
 - Pode ser usado com algumas propriedades (FOUND, NOTFOUND, ROWCOUNT). Esse último fornece quantas linhas foram afetadas pelo último comando DML utilizado antes do IF.
- ✓ Exemplo utilizando: **SQL%FOUND**

BEGIN

Update FUNC set VL_SAL=240 where VL_SAL < 200;

IF **SQL%FOUND** THEN

commit;

ELSE

dbms_output.put_line('Valores nao encontrados');

END IF;

END;

PL/SQL – Cursor Implícito

- Através dos atributos do cursor implícito (chamado SQL) podemos testar a saída produzida por um comando SQL:
 - ✓ **SQL%ROWCOUNT** - nº de linhas afetadas pelo último comando DML executado
 - ✓ **SQL%FOUND** - TRUE se o último comando DML afetou uma ou mais linhas
 - ✓ **SQL%NOTFOUND** - TRUE se o último comando DML não afetou nenhuma linha
 - ✓ **SQL%ISOPEN** - Sempre FALSE porque o PL/SQL fecha sempre os cursores implícitos depois de os executar;

PL/SQL – Cursor Explícito

- Os cursores explícitos são chamados dessa forma porque são declarados formalmente na área de declarações do módulo, ao contrário do que ocorre com os cursores implícitos. São tipicamente mais flexíveis e poderosos que os cursores implícitos, podendo substituí-los em qualquer situação.
- São utilizados para execução de consultas que possam retornar nenhuma ou mais de uma linha. Neste caso, o cursor deve ser explicitamente declarado na área **Declare**.
- O nome do cursor não pode ser igual ao da tabela.

✓ Exemplo:

DECLARE

CURSOR <nome do cursor> **IS** <declaração SELECT>;

BEGIN

.

END;

PL/SQL – Cursor Explícito

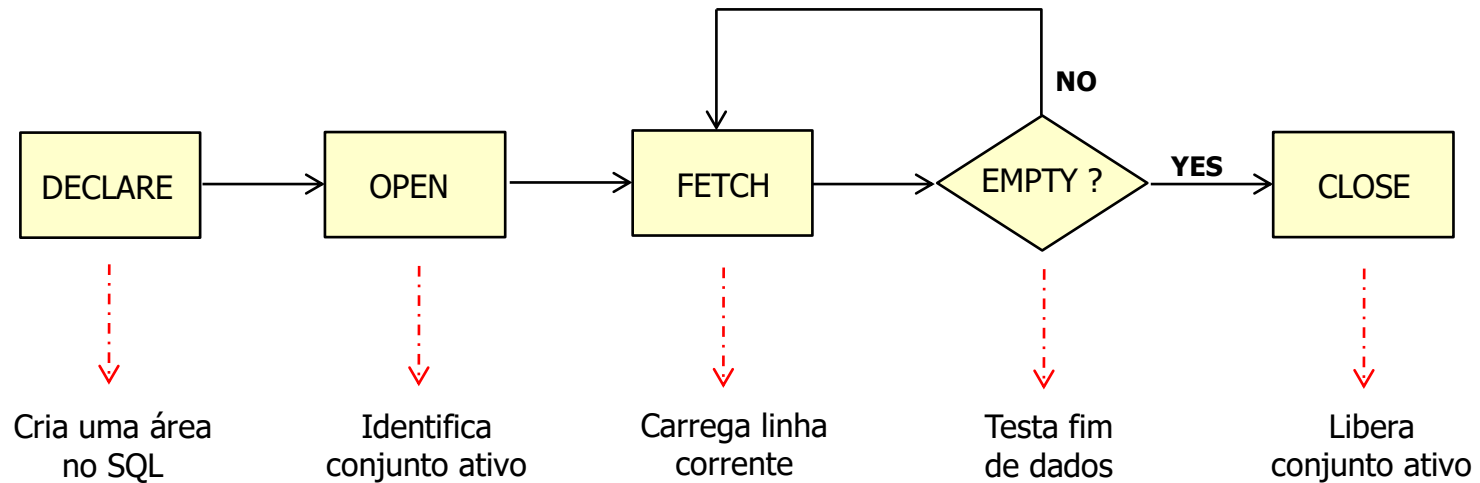
- Dentro de um cursor, o comando **SELECT** *não utiliza* a cláusula **INTO**.
- Ao fazermos a declaração, apenas foi definida uma estrutura que será utilizada posteriormente, quando o SELECT for executado e a recuperação das linhas for realizada.
- A primeira instrução realmente executável relativa ao cursor é a sua *abertura*, feita na área de instruções através do comando **OPEN**.
- É no momento da abertura que o **SELECT** é executado e as linhas recuperadas tornam-se disponíveis para uso.

PL/SQL – Cursor Explícito

- Para sua utilização são necessários alguns passos básicos:
 - ✓ declarar o cursor;
 - ✓ declarar as variáveis que receberão os dados;
 - ✓ abrir (uma espécie de preparação) o cursor na área de instruções;
 - ✓ ler os dados produzidos pelo cursor;
 - ✓ fechar (desalocar a memória) do cursor.

OPEN	Abre o cursor
FETCH	Disponibiliza a linha corrente e posiciona na próxima linha do cursor. As linhas armazenadas no cursor somente poderão ser processadas quando o seu conteúdo for transferido para variáveis que possam ser manipuladas no PL/SQL.
CLOSE	Fecha o cursor.

PL/SQL – Cursor Explícito



PL/SQL – Cursor Explícito

- ❑ Após declarar uma variável como sendo do tipo `nome_do_cursor%rowtype`, essa variável será um tipo de registro (variável composta de diversas subvariáveis) cujas subvariáveis terão os mesmos nomes, tipos e tamanhos e estarão na mesma ordem dos campos especificados no comando `SELECT` do cursor. O conteúdo da variável desse tipo é referenciado com `nome_do_registro.nome` da subvariável.
- ❑ Para cada cursor, quatro atributos podem ser verificados, e seus valores podem ser alterados a cada execução de um comando `FETCH`. Esses atributos são:

nome_cursor%FOUND	retorna TRUE caso <code>FETCH</code> consiga retornar alguma linha e FALSE caso contrário. Se nenhum <code>FETCH</code> tiver sido executado, será retornado NULL .
nome_cursor%NOTFOUND	retorna FALSE caso <code>FETCH</code> consiga retornar alguma linha e TRUE caso contrário. Se nenhum <code>FETCH</code> tiver sido executado, será retornado NULL .
nome_cursor%ROWCOUNT	retorna o número de linhas já processadas pelo cursor. Se nenhum <code>FETCH</code> tiver sido executado, será retornado 0 (zero).
nome_cursor%ISOPEN	retorna TRUE caso o cursor esteja aberto e FALSE caso contrário.

PL/SQL – Cursor Explícito

- O comando FETCH recupera a próxima linha de um cursor e pode ser utilizado com *variáveis* ou *registro*

✓ **FETCH** nome_cursor **INTO** var1, var2, ...;

Ou

✓ **FETCH** nome_cursor **INTO** reg_declarado;

onde:

- ✓ **var1, var2** – são as variáveis que vão receber os valores recuperados pelo cursor. Estas variáveis devem ser previamente declaradas e devem concordar em número e tipo com as expressões do comando SELECT do cursor.
- ✓ **reg_declarado** – nome de registro pré-declarado compatível com o cursor que irá receber os valores recuperados. Esse registro pode ser declarado da seguinte forma:

reg_declarado nome_cursor%ROWTYPE;

PL/SQL – Cursor Explícito

- ✓ Exemplo 1: Utilizando **cursor** com **registro**

DECLARE

CURSOR c_cliente **IS** SELECT codigo, nome FROM cliente;
v_cliente c_cliente%rowtype;

BEGIN

OPEN c_cliente;

LOOP

FETCH c_cliente **INTO** v_cliente;

EXIT when c_cliente%notfound;

DBMS_OUTPUT.PUT_LINE('Cliente: '||v_cliente.nome);

END LOOP;

CLOSE c_cliente;

END;

Obs: Convém fechar todo cursor. Máximo de cursores abertos (50).

PL/SQL – Cursor Explícito

- ✓ Exemplo 2: Utilizando **cursor** com **variáveis**

DECLARE

```
CURSOR c_cliente IS SELECT codigo, nome FROM cliente;  
v_codcli      cliente.codigo%type;  
v_nomecli     cliente.nome%type;
```

BEGIN

```
OPEN c_cliente;  
LOOP  
    FETCH c_cliente INTO v_codcli, v_nomecli;  
    EXIT when c_cliente%notfound;  
    DBMS_OUTPUT.PUT_LINE('Cliente: '||v_nomecli);  
END LOOP;  
CLOSE c_cliente;
```

END;

Nota: A exceção **NO_DATA_FOUND** é levantada apenas em instruções **SELECT...INTO**, quando a cláusula WHERE da consulta não corresponde a nenhuma linha. Quando nenhuma linha é retornada por um **cursor explícito** ou uma instrução UPDATE ou DELETE, o atributo **%NOTFOUND** é configurado como TRUE, não levantando a exceção **NO_DATA_FOUND**.

PL/SQL – Cursor Explícito

- O comando **FOR ... LOOP**, quando aplicado a um cursor, executa automaticamente as seguintes ações:
 - Cria a variável do tipo registro que receberá os dados (**DECLARE**);
 - Abre (**OPEN**) o cursor;
 - Copia as linhas uma a uma (**FETCH**), a cada interação do comando;
 - Controla o final do cursor;
 - Fecha (**CLOSE**) o cursor.

```
FOR <nome do registro> IN <nome do cursor> LOOP  
... ;  
END LOOP ;
```

NOTA: Caso seja necessário sair do loop do comando FOR durante sua execução, o cursor deverá ser fechado explicitamente com o comando CLOSE.

PL/SQL – Cursor Explícito

✓ Exemplo:

DECLARE

CURSOR c_cliente **IS** SELECT codigo, nome FROM cliente;

BEGIN

FOR v_cliente **IN** c_cliente **LOOP**

DBMS_OUTPUT.PUT_LINE('Cliente: '||v_cliente.nome);

END LOOP;

END;

PL/SQL – Cursor Explícito

- **Parâmetros de cursor** - Geralmente o comando SELECT de um cursor possui uma cláusula WHERE que especifica uma seleção de linhas a serem retornadas. Muitas vezes, temos necessidade de variar um dado a ser comparado nessa cláusula, e isso pode ser feito através de uma espécie de parâmetro passado para o cursor no momento de sua abertura.

✓ **Exemplo:**

DECLARE

CURSOR c_emp (**p_Dept** emp.deptno%TYPE, **p_Job** emp.job%TYPE) **IS**

SELECT * FROM emp WHERE deptno = p_Dept and job = p_Job;

v_emp c_emp%rowtype;

BEGIN

OPEN c_emp(**30**, '**SALESMAN**');

LOOP

FETCH c_emp **INTO** v_emp;

EXIT when c_emp%notfound;

DBMS_OUTPUT.PUT_LINE('Nome Empreg: ' || v_emp.ename);

END LOOP;

CLOSE c_emp;

END;

PL/SQL – Cursor Explícito

- **Parâmetros de cursor** – Outra forma de tornar o cursor mais flexível é pedir os valores dos parâmetros em tempo de execução, ao invés de fixá-los na abertura do cursor.

✓ **Exemplo:**

DECLARE

CURSOR c_emp (**p_Dept** emp.deptno%TYPE, **p_Job** emp.job%TYPE) **IS**

SELECT * FROM emp WHERE deptno = p_Dept and job = p_Job;

v_emp c_emp%rowtype;

BEGIN

OPEN c_emp(&p_Dept, '&p_Job');

LOOP

FETCH c_emp **INTO** v_emp;

EXIT when c_emp%notfound;

DBMS_OUTPUT.PUT_LINE('Nome Empreg: ' || v_emp.ename);

END LOOP;

CLOSE c_emp;

END;

PL/SQL – PROCEDURE

- Há dois tipos principais de blocos PL/SQL:
 - ✓ **Anônimo:** Bloco PL/SQL iniciado com DECLARE ou BEGIN. Não é armazenado diretamente no banco de dados e nem pode ser chamado diretamente de outros blocos PL/SQL.

```
[DECLARE]  
  
BEGIN  
  
--statements  
  
[EXCEPTION]  
  
END;
```

- ✓ **Identificado:** Bloco PL/SQL armazenado no banco de dados e executado quando apropriado. Referem-se as seguintes construções: procedures, functions, packages e triggers.

```
CREATE PROCEDURE name  
  
IS  
  
BEGIN  
  
--statements  
  
[EXCEPTION]  
  
END;
```

PL/SQL – PROCEDURE

- ❑ PROCEDURES são subprogramas que executam uma determinada ação. Não retornam valores, portanto, não são utilizadas para atribuir valores a variáveis ou como argumento em um comando SELECT.
- ❑ Devemos utilizar a instrução **CREATE OR REPLACE PROCEDURE** para criar ou substituir uma *procedure* (*neste último caso a procedure anterior é descartada sem nenhum aviso prévio*).
- ❑ Se a procedure existir e as palavras-chave **OR REPLACE** não estiverem presentes, a instrução CREATE retornará o erro Oracle:
“ORA-955: Nome já está sendo usado por outro objeto.”
- ❑ A procedure será então compilada e armazenada no banco de dados.
- ❑ O código-fonte e a forma compilada ficam armazenados no servidor de banco de dados.

PL/SQL – PROCEDURE

✓ Sintaxe:

CREATE [OR REPLACE] PROCEDURE nome_procedure

[(argumento1 modo tipo_de_dados,
argumento2 modo tipo_de_dados,
...
argumentoN modo tipo_de_dados)]

IS ou **AS**

[variáveis locais, constantes, ...]

BEGIN

...

END [nome_procedure];

/

} *corpo da procedure*

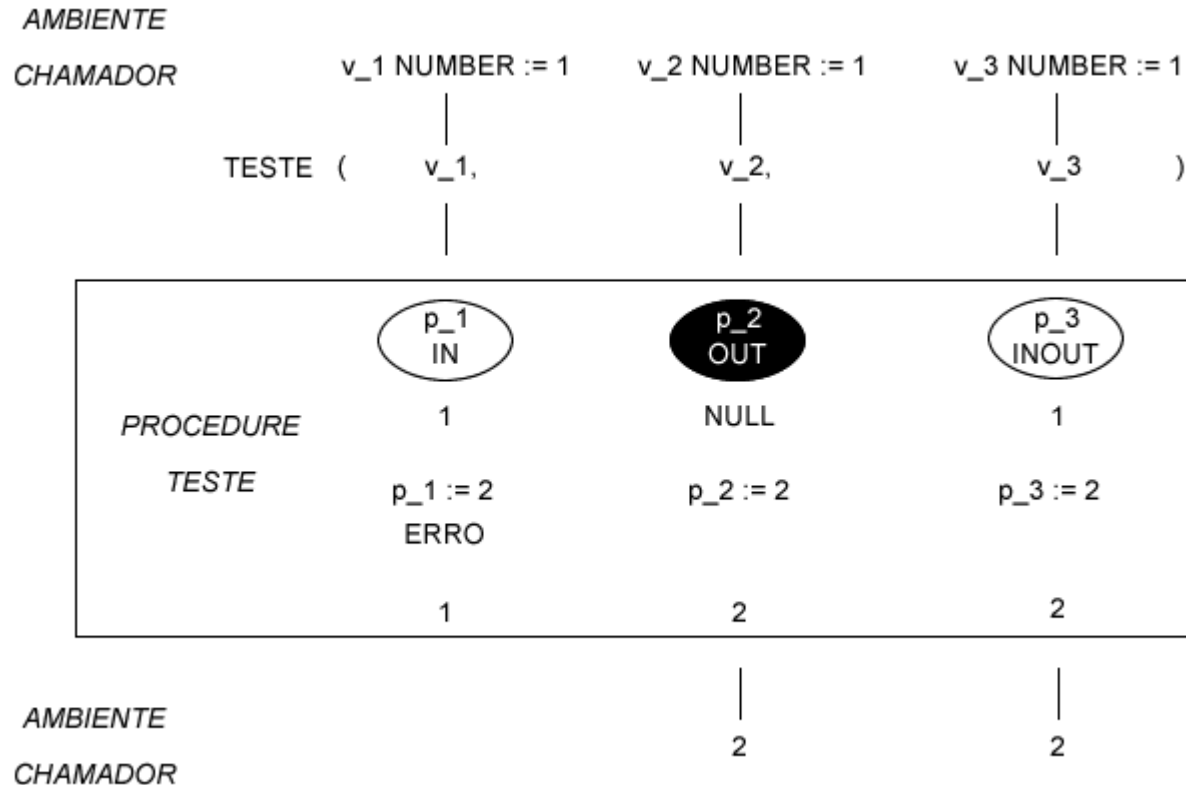
PL/SQL – PROCEDURE

- ❑ **Argumento** é o nome de uma variável PL/SQL passada para a procedure;
- ❑ **Modo** identifica o tipo de argumento (**IN**, **OUT** ou **IN OUT**);
- ❑ **Tipo de dado** corresponde ao datatype;
- ❑ **IS ou AS** → essas cláusulas são equivalentes, pode-se utilizar tanto uma quanto outra;
- ❑ **Bloco PL/SQL** é o corpo da procedure que define as ações que serão executadas quando a procedure for executada;
- ❑ A palavra **DECLARE** não é mais utilizada. As variáveis são declaradas imediatamente após a palavra AS (ou IS) que encerra a linha de definição do nome da "procedure". Caso não seja necessário declarar variáveis, a palavra BEGIN virá logo após o AS.

PL/SQL – PROCEDURE

- **Argumento** - Nome da variável que será enviada ou retornada do ambiente chamador para a procedure. Pode ser passado em um dos três modos a seguir:
 - ✓ **IN** (padrão): Passa um valor do ambiente chamador para procedure e este valor não pode ser alterado dentro da mesma.
 - ✓ **OUT**: Passa um valor da procedure para o ambiente chamador. Seu valor inicial dentro da "procedure" é NULL.
 - ✓ **IN OUT**: Passa um valor do ambiente chamador para a procedure. Esse valor pode ser alterado dentro da mesma e retornar com o valor atualizado para o ambiente chamador.
- Se o modo não for especificado, o parametro **IN** é adotado como padrão.
- O tipo do parâmetro, sem especificação de tamanho são válidos (por exemplo, NUMBER ou VARCHAR2), enquanto NUMBER(2) ou VARCHAR2(10) não são aceitos.

PL/SQL – PROCEDURE



PL/SQL – PROCEDURE

- ❑ **Corpo da procedure** - O corpo de uma procedure é um bloco PL/SQL com seções executáveis declarativas e de exceção.
 - ✓ A seção declarativa está localizada entre as palavras-chave **IS** ou **AS** e a palavra-chave **BEGIN**.
 - ✓ A seção executável (a **única requerida**) está localizada entre as palavras-chave **BEGIN** e **EXCEPTION** ou entre as palavras-chave **BEGIN** e **END**, caso não exista nenhuma seção de tratamento de exceções.
 - ✓ Se a seção de exceção (**EXCEPTION**) estiver presente, ela é colocada entre as palavras-chave **EXCEPTION** e **END**.

Exemplo:

```
CREATE OR REPLACE PROCEDURE nome_da_procedure [lista_de_parametros] AS  
/* A seção declarativa entra aqui*/  
BEGIN  
/* A seção executável entra aqui*/  
EXCEPTION  
/* A seção de exceção entra aqui*/  
END [nome_da_procedure];
```

PL/SQL – PROCEDURE

- ❑ Criando uma procedure com o argumento **IN**

- ✓ Armazenar dados de um novo empregado

CREATE OR REPLACE PROCEDURE *novos_empregados*

```
(v_emp_no      IN      emp.empno%TYPE,  
  v_emp_name   IN      emp.ename%TYPE,  
  v_emp_job     IN      emp.job%TYPE,  
  v_emp_hiredate IN      emp.hiredate%TYPE,  
  v_emp_sal     IN      emp.sal%TYPE,  
  v_dept_no     IN      emp.deptno%TYPE)
```

IS

BEGIN

```
INSERT INTO emp (empno, ename, job, hiredate, sal, deptno )
```

```
VALUES
```

```
(v_emp_no,    v_emp_name,    v_emp_job,    v_emp_hiredate,    v_emp_sal,  
  v_dept_no);
```

```
COMMIT;
```

END *novos_empregados*;

/

PL/SQL – PROCEDURE

- A execução do CREATE PROCEDURE apenas compila e armazena o procedimento no banco. Para executá-lo no SQL*Plus digite o comando:

EXECUTE nome_da_procedure (**argum1**, **argumN**)

- Executando a procedure *NOVOS_EMPREGADOS* a partir da linha de comando:

```
SQL> EXEC NOVOS_EMPREGADOS (1,JOÃO, GERENTE, 10/05/2011,  
5600, 30)
```

Nota: Os argumentos são todos aqueles citados na procedure entre o comando CREATE PROCEDURE e a cláusula IS. Exemplo:

```
CREATE OR REPLACE PROCEDURE novos_empregados  
  (v_emp_no           IN      emp.empno%TYPE,  
   v_emp_name        IN      emp.ename%TYPE,  
   v_emp_job         IN      emp.job%TYPE,  
   v_emp_hiredate    IN      emp.hiredate%TYPE,  
   v_emp_sal         IN      emp.sal%TYPE,  
   v_dept_no        IN      emp.deptno%TYPE)  
IS  
  :  
  :
```

PL/SQL – PROCEDURE

- ❑ Criando uma procedure para devolver informações sobre um empregado.

```
CREATE OR REPLACE PROCEDURE pesquisa_empregado
```

```
    ( v_emp_no      IN          emp.empno%TYPE)
```

```
IS
```

```
    v_emp_name      emp.ename%TYPE;
```

```
    v_emp_sal       emp.sal%TYPE;
```

```
    v_emp_comm      emp.comm%TYPE;
```

```
BEGIN
```

```
    SELECT   ename,   sal,   comm   INTO   v_emp_name,   v_emp_sal,  
            v_emp_comm FROM emp WHERE empno = v_emp_no;
```

```
    DBMS_OUTPUT.PUT_LINE('Nome:   ' || v_emp_name || ' - Salario: ' ||  
v_emp_sal || ' - Comissao: ' || v_emp_comm);
```

```
END pesquisa_empregado;
```

```
/
```

PL/SQL – PROCEDURE

- ❑ Executando a procedure *PESQUISA_EMPREGADO* da página anterior através de um bloco PL/SQL

DECLARE

v_emp emp.empno%type := &v_emp;

BEGIN

pesquisa_empregado(v_emp);

END;

/

Nota: Para executar uma procedure dentro de um bloco PL/SQL, escreva apenas seu nome seguido dos argumentos dentro do parênteses, conforme exemplo acima. O comando EXEC só é utilizado quando a procedure é executada na linha de comando do SQL/Plus.

PL/SQL – PROCEDURE

- ❑ Criando uma procedure com o argumento **IN OUT**
- ✓ Transforme a seqüência de sete dígitos de um número para o formato de número de telefone.

```
CREATE OR REPLACE PROCEDURE formata_telefone
```

```
    (v_phone_no    IN OUT  varchar)
```

```
IS
```

```
BEGIN
```

```
    v_phone_no := SUBSTR (v_phone_no,1,4) || '-' || SUBSTR (v_phone_no,5,4);
```

```
    Dbms_output.put_line('Telefone: ' || v_phone_no);
```

```
END formata_telefone;
```

```
/
```

Nota: Procedures com argumentos do tipo **OUT** ou **IN OUT** só poderão ser chamadas de dentro de um bloco PL/SQL, **nunca** pela linha de comando com o comando **EXEC**.

PL/SQL – PROCEDURE

- Executar a procedure ***FORMATATA_TELEFONE*** da página anterior, através de um bloco PL/SQL:

DECLARE

v_phone_no varchar(15) := '&v_phone_no';

BEGIN

formatata_telefone (v_phone_no);

END;

/

PL/SQL – PROCEDURE

✓ Exemplo 1:

- Criar a tabela **PRODUTO** e inserir linhas.

```
CREATE TABLE produto (codigo NUMBER(4) primary key, nome VARCHAR(20),  
valor NUMBER(7,2), categoria NUMBER(4));
```

```
INSERT INTO produto VALUES (1,'produto1',2.5,10);
```

```
INSERT INTO produto VALUES (2,'produto2',3.2,20);
```

```
INSERT INTO produto VALUES (3,'produto3',5.8,30);
```

- Criar a procedure **AUMENTA_VALOR** para atualizar dados na tabela **PRODUTO**:

```
CREATE OR REPLACE PROCEDURE aumenta_valor
```

```
(v_categoria          IN          produto.categoria%TYPE,  
v_percentual          NUMBER)
```

```
IS
```

```
BEGIN
```

```
    UPDATE produto SET valor = valor * (1 + (v_percentual / 100))  
    WHERE categoria = v_categoria;
```

```
END aumenta_valor;
```

```
/
```

PL/SQL – PROCEDURE

- Executar a procedure **AUMENTA_VALOR** , da página anterior, passando os argumentos 10 e 8 para Código da Categoria e Percentual, respectivamente:

EXEC aumenta_valor (10 , 8);

- Visualização da tabela PRODUTO após execução da procedure AUMENTA_VALOR, atribuindo um aumento de 8% sobre os produtos da categoria 10.

SELECT * FROM produto;

CODIGO	NOME	VALOR	CATEGORIA
-----	-----	-----	-----
1	produto1	54,00	10
2	produto2	120,00	20
3	produto3	80,00	30

PL/SQL – PROCEDURE

- ✓ **Exemplo 2:** Cria uma procedure que recebe dois valores inteiros p_1 e p_2 e imprime na tela o total (atribuído ao terceiro parâmetro p_t)

```
CREATE OR REPLACE PROCEDURE soma
(
    p_1 IN NUMBER, p_2 IN NUMBER, p_t OUT NUMBER
)
IS
BEGIN
    p_t := p_1 + p_2;
    DBMS_OUTPUT.PUT_LINE(p_1 || ' + ' || p_2 || ' = ' || p_t);
END soma;
/
```

Nota: Observe acima que os parâmetros p_1 e p_2 foram definidos com o modo **IN**, pois representam valores que *não serão* alterados dentro da *procedure*. Já o parâmetro p_t foi definido com o modo **OUT**, pois o valor *será alterado* no corpo da *procedure*.

PL/SQL – PROCEDURE

- Ainda sobre o exemplo da página anterior, outro detalhe a ser observado é o nome da procedure (**SOMA**) depois da instrução **END**. Sua utilização, embora opcional, é recomendável, pois torna a procedure mais fácil de ler e permite que o compilador sinalize pares BEGIN-END que não se correspondem, logo que possível.
- A procedure SOMA poderá ser chamada por um bloco PL/SQL anônimo. Neste caso, escreva apenas o nome da procedure seguida de seus parâmetros (argumentos).

DECLARE

```
v_1      NUMBER(2) := 4;  
v_2      NUMBER(2) := 6;  
v_t      NUMBER(2)  ;
```

BEGIN

```
SOMA (v_1, v_2, v_t);
```

```
END;
```

```
/
```

Nota: Se não houver nenhum parâmetro em uma procedure, não se deve utilizar parênteses na declaração, nem na chamada da procedure.

PL/SQL – PROCEDURE

- Visualizar erros de compilação de uma procedure durante sua criação:

SHOW ERRORS PROCEDURE *nome_procedure*

- Visualizar o código-fonte de uma procedure:

SELECT TEXT FROM USER_SOURCE
WHERE NAME = '*nome_da_procedure*' ;

- Para eliminar uma procedure:

DROP PROCEDURE *nome_da_procedure*;

- Assim como outras instruções CREATE, criar uma procedure é uma operação de DDL, portanto, um COMMIT implícito é feito após a execução do comando *CREATE PROCEDURE*.

PL/SQL – FUNCTION

- ❑ FUNCTIONS são subprogramas que executam uma determinada ação e retornam valores.
- ❑ Além das funções pré-definidas, como TO_CHAR, UPPER, ROUND, etc, podemos definir nossas próprias funções com códigos PL/SQL e utilizá-las em outros códigos ou mesmo dentro de comandos SQL padrão, como o SELECT.
- ❑ Toda FUNCTION **retorna valor**. Caso seja feita uma FUNCTION sem o RETURN, o SQL/PLUS apresentará erro.
- ❑ A FUNCTION retornar apenas um valor. Se for necessário retornar vários valores, substitua a FUNCTION por uma PROCEDURE com argumentos do tipo OUT.

PL/SQL – FUNCTION

- ❑ As funções tornam o banco mais lento, pois, cada vez que são chamadas são executadas e devolvem valor, consumindo memória e processador.
- ❑ Dependendo do que se deseja, a PROCEDURE é mais indicada e o banco trabalha melhor.
- ❑ Se uma função for utilizada para retornar como uma determinada coluna de um comando select, ela será chamada exatamente a quantidade de vezes correspondente ao número de registros da tabela. Se a tabela tiver 1.000.000 de registros, esta função será executada 1.000.000 de vezes.

PL/SQL – FUNCTION

✓ Sintaxe:

```
CREATE [OR REPLACE] FUNCTION nome_funcao  
  [(argumento1 modo tipo_de_dados,  
   argumento2 modo tipo_de_dados,  
   ...  
   argumentoN modo tipo_de_dados)]  
RETURN tipo_de_dados  
IS ou AS  
  [variáveis locais, constantes, ...]  
BEGIN  
  ...  
END [nome_funcao];  
/
```

PL/SQL – FUNCTION

- ✓ **Exemplo:** Criar uma função que recebe um número de RA de aluno como uma entrada e retorna o nome e o sobrenome concatenados.
- *Criar a tabela ALUNO e inserir os registros antes de criar a FUNÇÃO.*

```
CREATE TABLE ALUNO (RA NUMBER, NOME VARCHAR2(20),  
SOBRENOME VARCHAR2(30));
```

```
INSERT INTO ALUNO VALUES (1, 'ANTONIO', 'ALVES');
```

```
INSERT INTO ALUNO VALUES (2, 'BEATRIZ', 'BERNARDES');
```

PL/SQL – FUNCTION

- Criar a função *NOME_ALUNO*

```
CREATE OR REPLACE FUNCTION NOME_ALUNO (P_RA ALUNO.RA%TYPE)
RETURN VARCHAR
IS
    V_NOMECompleto VARCHAR(60);
BEGIN
    SELECT NOME || ' ' || SOBRENOME INTO V_NOMECompleto  FROM ALUNO
        WHERE RA = P_RA;
RETURN V_NOMECompleto;
END NOME_ALUNO;
/
```

- Executar a função através de um comando SELECT:

```
SELECT RA, NOME_ALUNO(RA) "NOME COMPLETO" FROM ALUNO;
```

<u>RA</u>	<u>NOME COMPLETO</u>
1	ANTONIO ALVES
2	BEATRIZ BERNARDES

PL/SQL – FUNCTION

- Visualizar erros de compilação de uma função durante sua criação:

SHOW ERRORS FUNCTION *nome_funcao*

- Visualizar o código-fonte de uma função:

SELECT TEXT **FROM** USER_SOURCE
WHERE NAME = '*nome_da_funcao*';

- Para eliminar uma função:

DROP FUNCTION *nome_da_funcao*;

- Assim como outras instruções CREATE, criar uma função é uma operação de DDL, portanto, um COMMIT implícito é feito após a execução do comando *CREATE FUNCTION*.

PL/SQL – PROCEDURE x FUNCTION

COMPARANDO PROCEDURE x FUNCTION

PROCEDURE	FUNCTION
Executa como uma instrução PL/SQL.	Chamada como parte de uma expressão.
Não pode ser usada em uma instrução SELEC.	Pode ser usada em uma instrução SELECT.
Não pode retornar dado com instrução RETURN.	Tem de conter uma instrução RETURN para retornar dado.
Pode ter parâmetro de entrada / saída.	Pode ter apenas parâmetro de entrada.
Pode retornar nenhum, um, ou muitos valores	Tem de retornar um valor único
Não pode ser chamada a partir de Function.	Pode ser chamada de dentro de Procedure.

PL/SQL – TRIGGER

- ❑ TRIGGERS (gatilhos) são blocos PL/SQL disparados automática e implicitamente sempre que ocorrer um evento associado a uma tabela (INSERT, UPDATE ou DELETE).
- ❑ Utilizadas para:
 - ✓ Manutenção de tabelas
 - ✓ Implementação de níveis de segurança mais complexos
 - ✓ Geração de valores de colunas
(Exemplo: gerar o valor total do pedido a cada inclusão, alteração ou exclusão na tabela item_pedido)

PL/SQL – TRIGGER

✓ Sintaxe:

```
CREATE [OR REPLACE] TRIGGER nome_trigger  
    {BEFORE/AFTER} {INSERT,UPDATE,DELETE} OF (nome_coluna1,  
    nome_coluna2, ...) ON nome_tabela  
FOR EACH ROW  
REFERENCING, OLD AS ANTIGO NEW AS NOVO  
WHEN condição  
DECLARE  
...  
BEGIN  
...  
END;
```

NOTA: A cláusula REFERENCING está substituindo as áreas de memória OLD e NEW por ANTIGO e NOVO.

PL/SQL – TRIGGER

▣ **TEMPO**

Os tempos de uma trigger podem ser:

- ✓ BEFORE - antes do evento
- ✓ AFTER - depois do evento

▣ **EVENTO**

Os eventos de uma trigger podem ser:

- ✓ INSERT
- ✓ UPDATE
- ✓ DELETE

PL/SQL – TRIGGER

- **TIPO** - Indica quantas vezes a trigger poderá ser disparada. Os tipos podem ser:
 - ✓ **Comando**: acionada antes ou depois de um comando, independentemente deste afetar uma ou mais linhas.
 - Não permite acesso as linhas atualizadas por meio dos prefixos :OLD e :NEW.
 - Não utiliza a cláusula FOR EACH ROW no cabeçalho de criação.
 - ✓ **Linha**: acionada uma vez para cada linha afetada pelo comando ao qual a trigger estiver associada.
 - Permite o uso dos prefixos :OLD e :NEW no corpo da trigger e das cláusulas REFERENCING e WHEN em seu cabeçalho.
 - Deve-se incluir a cláusula FOR EACH ROW no cabeçalho.
- **Cláusula WHEN** - Utilizada para restringir as linhas que irão disparar a trigger.

PL/SQL – TRIGGER

- ✓ Exemplo: Criar uma trigger que adicione registros na tabela VALOR_PRODUTO sempre que uma linha for atualizada na tabela PRODUTO ao ser executado o comando UPDATE.
- Criar as tabelas PRODUTO e VALOR_PRODUTO antes de executar a trigger.

```
CREATE TABLE PRODUTO
```

```
(codigo NUMBER(4), valor NUMBER(7,2));
```

```
CREATE TABLE VALOR_PRODUTO
```

```
(codigo NUMBER(4), valor_anterior NUMBER(7,2),  
valor_novo NUMBER(7,2));
```

PL/SQL – TRIGGER

- ❑ Criando a trigger **VERIFICA_VALOR**:

```
SQL> CREATE OR REPLACE TRIGGER verifica_valor  
  BEFORE UPDATE OF valor ON Produto  
  FOR EACH ROW  
  BEGIN  
    INSERT INTO valor_produto  
      (codigo, valor_anterior, valor_novo)  
    VALUES  
      (:OLD.codigo, :OLD.valor, :NEW.valor);  
  END;  
  /
```

PL/SQL – TRIGGER

- ❑ Inserindo registros na tabela **PRODUTO**:

```
INSERT INTO produto VALUES (1,2.5);  
INSERT INTO produto VALUES (2,3.2);  
INSERT INTO produto VALUES (3,5.8);
```

- ❑ Atualizando dados na tabela PRODUTO. Neste momento, antes do comando UPDATE ser concluído, a trigger será disparada e um novo registro será inserido na tabela VALOR_PRODUTO para armazenar o valor antigo e o novo valor do produto mencionado na cláusula WHERE do comando UPDATE, conforme corpo da trigger na página anterior:

```
UPDATE produto SET valor = 5.4 WHERE codigo = 3;
```

- ❑ Visualizando dados na tabela VALOR_PRODUTO:

```
SELECT * FROM valor_produto;
```

<u>CODIGO</u>	<u>VALOR_ANTERIOR</u>	<u>VALOR_NOVO</u>
3	5.8	5.4

PL/SQL – TRIGGER

▣ Regras para criação de triggers:

- ✓ Número máximo de triggers possíveis para uma tabela (**12**), podendo ser um de cada tipo (BEFORE UPDATE <row>, BEFORE DELETE <row>, BEFORE INSERT <row>, BEFORE INSERT <comando>, BEFORE UPDATE <comando>, BEFORE DELETE <comando> e as mesmas sintaxes para AFTER).
- ✓ Não podem ser utilizados os comandos COMMIT e ROLLBACK, inclusive em procedures e functions chamadas pela trigger.
- ✓ Não podem ser alteradas chaves primárias, únicas ou estrangeiras.
- ✓ Não podem ser feitas referências a campos do tipo LONG E LONG RAW.

PL/SQL – TRIGGER

Conteúdo das áreas OLD e NEW (apenas triggers de linha)

EVENTO	OLD	NEW
INSERT	NULL	Valor inserido
UPDATE	Valor antes da alteração	Valor após a alteração
DELETE	Valor antes da exclusão	NULL

- ❑ Em triggers com cláusula de tempo BEFORE é possível consultar e alterar o valor de :NEW.
- ❑ Em triggers com cláusula de tempo AFTER é possível apenas consultar o valor de :NEW.

PL/SQL – TRIGGER

- Para testar o evento de chamada da trigger são disponibilizados os seguintes predicados:
 - ✓ **Inserting**: retorna TRUE se a trigger foi disparada por um comando INSERT.
 - ✓ **Updating**: retorna TRUE se a trigger foi disparada por um comando UPDATE.
 - ✓ **Deleting**: retorna TRUE se a trigger foi disparada por um comando DELETE.

PL/SQL – TRIGGER

✓ Exemplo:

```
CREATE OR REPLACE TRIGGER audit_emp
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
BEGIN
    IF DELETING THEN
        UPDATE audit_table SET del = del + 1
            WHERE user_name = user AND table_name = `EMP`
            AND column_name IS NULL;
    ELSIF INSERTING THEN
        UPDATE audit_table SET ins = ins + 1
            WHERE user_name = user AND table_name = `EMP`
            AND column_name IS NULL;
    ELSIF UPDATING THEN
        UPDATE audit_table SET upd = upd + 1
            WHERE user_name = user AND table_name = `EMP`
            AND column_name IS NULL;
    END IF;
END ;
```


PL/SQL – TRIGGER

- Para eliminar uma trigger:

DROP TRIGGER *nome_da_trigger*;

- Um trigger pode ser desativado sem ser excluído. Quando uma trigger é desativada, ela ainda existe no dicionário de dados, porém nunca é acionada.

ALTER TRIGGER *nome_da_trigger* **DISABLE**;

ALTER TRIGGER *nome_da_trigger* **ENABLE**;

- Todas as triggers para uma tabela particular também podem ser ativadas ou desativadas utilizando o comando ALTER TABLE.

ALTER TABLE *nome_da_tabela* **DISABLE ALL TRIGGERS**;

ALTER TABLE *nome_da_tabela* **ENABLE ALL TRIGGERS**;

- Views do dicionário de dados que contém informações sobre as triggers e seus status: **USER_TRIGGERS**, **ALL_TRIGGERS**, **DBA_TRIGGERS**.

PL/SQL – SQL DINÂMICO

- Um bloco PL/SQL não executa comandos DDL (CREATE TABLE, DROP TABLE, TRUNCATE, etc.), por isso, o Oracle oferece um recurso conhecido como NDS (Native Dynamic SQL) que permite, por meio da linguagem PL/SQL, executar dinamicamente comandos DDL.
- SQL dinâmico é um comando válido, codificado dentro de uma string e executado através do comando **EXECUTE IMMEDIATE**.
- A procedure abaixo utiliza um comando DDL (CREATE TABLE) e *não poderá ser criada*.

✓ Exemplo:

```
CREATE OR REPLACE PROCEDURE cria_tabela
IS
BEGIN
    create table teste (coluna1 number(5));
END cria_tabela;
```

PL/SQL – SQL DINÂMICO

- ▣ Para criá-la, devemos utilizar SQL dinâmico, conforme segue:

```
CREATE OR REPLACE PROCEDURE cria_tabela  
(nome_tabela IN VARCHAR2)  
IS  
    var_comando VARCHAR2(100);  
BEGIN  
    var_comando := 'create table '||nome_tabela||' (coluna1 number(5) )';  
    EXECUTE IMMEDIATE var_comando;  
END cria_tabela;  
/
```

- ▣ Para executar a procedure CRIA_TABELA:

```
EXEC cria_tabela ('teste');
```

PL/SQL – SQL DINÂMICO

- Inserindo linhas na tabela TESTE criada através da procedure CRIA_TABELA da página anterior:

```
INSERT INTO teste VALUES (1);
```

```
INSERT INTO teste VALUES (2);
```

```
INSERT INTO teste VALUES (3);
```

PL/SQL – SQL DINÂMICO

- Para eliminar a tabela também devemos utilizar SQL dinâmico:

```
CREATE OR REPLACE PROCEDURE limpa_teste
IS
    var_comando VARCHAR2(100);
BEGIN
    var_comando := 'TRUNCATE TABLE teste';
    EXECUTE IMMEDIATE var_comando;
END limpa_teste;
/
```

- Para executar a procedure LIMPA_TESTE:

```
EXEC limpa_teste;
```