

# Oracle PL/SQL

**Por**

**Edson Almeida Junior**



Consulting Tecnologia & Sistemas Ltda  
[www.consulting.com.br](http://www.consulting.com.br)

Março-2006

Última Atualização 11/04/2006

## 1. INTRODUÇÃO

### Introdução

---

Ao desenvolvermos um projeto temos em mente algumas questões para que o sistema implementando satisfaça as nossas necessidades de negócio dentro das nossas limitações de relação custo x benefício.

Como **Organizar** a implementação de um sistema?

Como fazer para que sua **Administração** seja simples ?

Como garantir um bom **Desempenho** ?

Os recursos disponíveis no RDMBS Oracle 10G são ferramentas que, com certeza, nos auxiliarão a obter nossos objetivos. Tudo que será aprendido aqui será compatível com o Oracle 8i, 9i e 10g.

### PL/SQL

---

Linguagem de procedimentos do Oracle. Através dela podemos criar diversas “rotinas” de programação, dentre elas, Procedimentos (Procedures), Gatilhos (Triggers) e Funções (Functions).

DECLARE

- Variáveis
- Cursores
- Procedures ou Funções

BEGIN

- Processamento

EXCEPTION

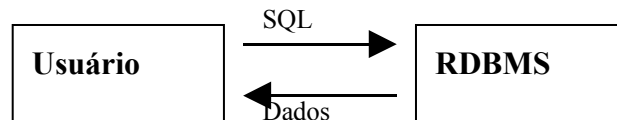
- Tratamento de Excessões

END;

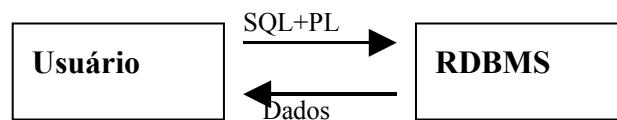
## Introdução

---

A primeira estrutura na qual um SGBD Relacional estava baseado trazia o SQL puro como única forma de interface com o banco.



O perfil de utilização dos bancos de dados relacionais foi evoluindo e essa estrutura, com o passar do tempo, já não respondia às necessidades. Agregou-se então extensões ao SQL que pretendiam responder às novas necessidades adicionando comandos procedurais a linguagem declarativa do SQL. No caso da Oracle essa extensão foi chamada de Procedural Language (PL/SQL).



## Introdução

---

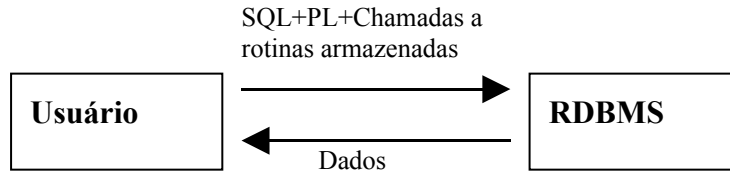
Recentemente o perfil de utilização evoluiu novamente e nos vemos diante de novas necessidades. Estruturas mais complexas como Banco de Dados Distribuídos, grande quantidade de informação, custo de tráfego em rede e Arquitetura Cliente-Servidor são algumas características desse novo perfil.

Na Arquitetura Cliente-Servidor, por exemplo, é muito importante manter o tráfego de rede a um mínimo e aumentar a performance do RDBMS de modo que ele possa atender rapidamente as freqüentes transações solicitadas pelo usuário.

Com essas especificações, organizar o processamento tornando-se fundamental. Por exemplo, grandes blocos PL/SQL circulando pela rede e sendo compilados e otimizados pelo servidor, a cada nova execução, não é uma boa solução. A melhor idéia é armazenar rotinas dentro do RDBMS e acionar essas rotinas através de chamadas. Em vez de todo um bloco PL/SQL temos circulando pela rede apenas o nome da rotina e seus parâmetros. Além disso o Oracle pode guardar a forma compilada e otimizada dessas rotinas melhorando o tempo de execução. No Oracle7 ou Oracle8 isto é um Procedimento Armazenado, ou **STORED PROCEDURE**.

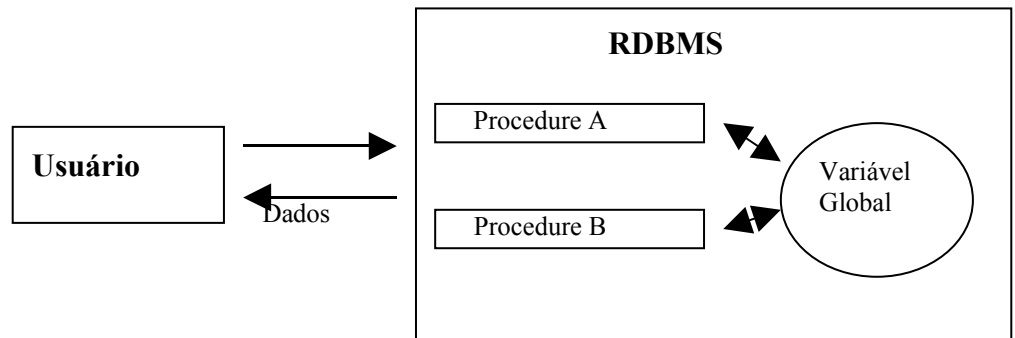
## Introdução

---



Além das vantagens de performance, o uso de Stored Procedures torna o sistema mais modular. As clássicas vantagens de um sistema estruturado: reaproveitamento de código, identificação dos pontos críticos do sistema e facilidade de manutenção ganham agora uma nova abordagem.

Neste sentido se torna interessante que o RDBMS suporte melhor o desenvolvimento do que simplesmente armazenar rotinas isoladas. É frequentemente necessário que diversas rotinas compartilhem as mesmas variáveis globais e que chamam rotinas não diretamente disponíveis para o usuário. No Oracle7 isso é suportado pelo uso de Pacotes, ou **PACKAGES**.

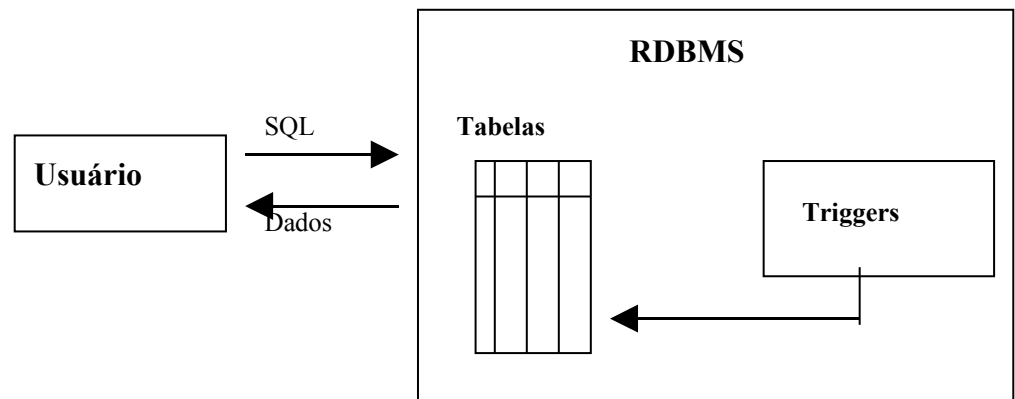


## Introdução

---

Uma evolução natural nessa tecnologia é tornar o banco de dados mais inteligente. O RDBMS tem o controle de todas as operações sobre os dados, se adicionarmos o conceito de Stored Procedures fica fácil imaginar o banco de dados monitorando determinados eventos e acionando rotinas automáticas.

Essas rotinas automáticas servem não só para garantir a integridade do banco de dados mas também garantir as regras inerentes ao negócio. Assim como temos uma chave primária ou uma restrição a um domínio ( constraints primary key e check implementadas pelo RDBMS ), podemos ter, no negócio, restrições que limitem uma venda a uma determinada porcentagem do estoque ou que determinem o limite de crédito de um cliente baseado no seu volume de negócio. Até hoje essas regras eram programadas dentro da aplicação. O programador tinha que se preocupar com elas ao mesmo tempo que criava as rotinas específicas da aplicação. Isso tornava o desenvolvimento mais lento, o código mais difícil de manter e aumentava a possibilidade de erros, além disso uma mudança nas regras de negócio implicava freqüentemente na alteração de vários programas. As rotinas automáticas do Oracle7 ou Oracle8 são chamadas de Gatilhos, ou **DATABASE TRIGGERS**.



## 2. Stored Procedures

### Criando Stored Procedures

---

#### Exemplo 1:

```
CREATE PROCEDURE PROCESSAMENTO
AS
BEGIN
    INSERT INTO AUDITORIA ( SYSDATE,
        'Termino Normal do Processamento ');
    COMMIT WORK;
END;
```

## Criando Stored Procedures

---

### Exemplo 2:

```

CREATE OR REPLACE PROCEDURE PROCESSAMENTO
IS
BEGIN
    INSERT INTO AUDITORIA (SYSDATE,
        'Termino Normal do Processamento ');
    COMMIT WORK;
EXCEPTION
    WHEN OTHERS THEN
        IF ..... THEN

            ELSE .....
        ROLLBACK;
END IF;
END PROCESSAMENTO;

```

## Executando Stored Procedures

---

### No SQLPLUS:

```
SQL> EXECUTE PROCESSAMENTO;
```

### No PL/SQL

```

DECLARE
    ( declaração de variáveis)
BEGIN
    ( Instruções)

PROCESSAMENTO;

(Instruções)

END;

```

### No SQLFORMS ( Dentro de um Trigger ou Procedure )

```

If: BLOCO10.UF = 'CE' Then
    PROCESSAMENTO;
Else
    ROLLBACK;
End If;

```

## Usando Parâmetros

---

Os nomes e tipos das variáveis usadas como parâmetros devem ser declarados quando a procedure é criada. Na especificação do tipo **NÃO** deverá constar o tamanho, mas declarações %TYPE e %ROWTYPE são permitidas.

Os valores são substituídos em termo de chamada da procedure, variando assim a cada execução.

Os parâmetros podem ser:

Apenas de Entrada:

**CREATE PROCEDURE TESTE ( var1 IN Varchar2 )**

ou

**CREATE PROCEDURE TESTE ( var1 Varchar2 )**

onde var1 é uma variável de entrada de dados para a procedure. A passagem nesse caso é feita por valor.

Esta é a modalidade **Default**.

Apenas de Saída:

**CREATE PROCEDURE TESTE1 ( var2 OUT Number )**

onde var2 é uma variável de retorno de dados da procedure para o ambiente chamador.

De Entrada e Saída:

**CREATE PROCEDURE TESTE3 ( var3 IN OUT Number )**

onde var3 é uma variável de compartilhamento de dados entre a procedure e o ambiente chamador. A passagem nesse caso é feita por referência.

## Usando Parâmetros

---

Os diversos tipos de parâmetros podem ser usados juntos em qualquer combinação:

OBS: Uma Stored Procedure nunca enxerga qualquer variável do ambiente chamador. Toda a interface entre o chamador e a Stored Procedure é feita através de parâmetros.

### Exemplo:

#### Criação

```
CREATE PROCEDURE
Aumento ( salário IN OUT Number,
          taxa IN Number,
          Status OUT Number )

AS
BEGIN
    salário: = salário * taxa;
    If salário > 5000 Then
        Status: = 1;
    Else
        Status: = 0;
    End If
END Aumento;
```

## Usando Parâmetros

---

Execução a partir de um bloco PL/SQL

```

DECLARE
    Cursor c1 Is
        Select id, salary
        From Emp
        for Update of salary;
    sal Emp.salary%TYPE;
BEGIN
    For rc1 In c1 Loop
        Sal := rc1.salary;
        Aumento(Sal,1.10);
        If sal < 1500 then
            Update Emp
            Set salary = sal
            Where Current of c1;
        End If;
    End Loop;
END;
```

## Exercícios

---

1. Crie uma tabela com o seguinte comando:

```

CREATE TABLE AUDITE
    ( tipo          Varchar2 (10),
    usuário Varchar2 (10),
    data           Date,
    texto         Varchar2(80);
```

Usando esta tabela crie a Stored Procedure: P\_Log ( tipo, texto ). Que inclui uma linha na tabela de log usando a data e usuário corrente.

2. Crie uma Stored Procedure que aumente os valores do salário do empregado passando o ID do empregado. A porcentagem de aumento deve ser passada como parâmetro. ( Use a P\_Log() )

Ex: P\_Aumento (10);

Log => ( 'AUMENTO', 'ALUNO1', '05-JUL-98', 'Aumento determinado pela direção da empresa' )

3. Crie uma Stored Procedure que verifique o salário de cada Empregado e os empregados que tiver o salário menor que 900 e o departamento NAME = 'Operations' promova um aumento de 35% no salário. Gerar na tabela AUDITE o log desta alterações.

Ex: P\_aumento1

```

Log => ( 'AUMENTO', 'SCOTT', '05-JUL-98',
        'Aumento no departamento Operations, por produtividade' )
```

Opcional:

4) Criar uma procedure que altera em 45% o salário dos empregados que são subordinados aos gerentes que não tem comissão.

### 3. Functions

#### Criando Functions

---

##### Exemplo:

```
CREATE OR REPLACE FUNCTION
Aumento ( salário Number )
RETURN Number;
AS

    aux Number;
BEGIN
    aux: = salario * 1.1 ;
    return(aux);
END;
```

#### Chamando Functions

---

##### No SQLPLUS:

Não é possível chamar uma função diretamente do prompt do SQLPLUS. Use um bloco PL/SQL.

##### No PL/SQL

```
DECLARE
    { declaração de variáveis}
BEGIN
    { Instruções }
    novosal := Aumento ( sal_func);
    {Instruções }
END;
```

##### No SQLFORMS ( Dentro de um Trigger ou Procedure)

```
If: BLOCO1.CAMPO1 = 'S' Then
    :BLOCO1NOVOSAL:=
Aumento ( : BLOCO1.SALFUNC);
Else
    ROLLBACK;
End If;
```

## Comparando Funções e Procedures

---

### Criando:

```
CREATE PROCEDURE
Aumento ( salent IN Number, salsaida OUT Number )
AS
BEGIN
    salsaida: = salent * 1.10;
END Aumento;
```

### Executando:

```
DECLARE
    salini      Number;
    salnov      Number;
BEGIN
    { Instruções }
    Aumento ( salini,salnov);
    { Instruções }
END;
```

### Criando:

```
CREATE FUNCTION
Aumento ( salent IN Number )
RETURN Number
AS
    aux Number;
BEGIN
    aux := salent * 1.10;
    Return (aux);
END Aumento;
```

### Executando:

```
DECLARE
    salini      Number;
    salnov      Number;
BEGIN
    { Instruções }
    salnov := Aumento( salini);
    { Instruções }
END;
```

## Exercícios

---

1.) Crie uma função que receba o número do telefone no formato 08199227401 (DDDFone) e devolva o número no formato (081)9922-7401.

Ex: P\_Fone (08199227401)

## 4. Tópicos Especiais em Functions e Stored Procedures

### Transações

---

As Functions e Stored Procedures podem incluir qualquer comando SQL ou PL/SQL, porém uma atenção especial deve ser dada aos comandos de controle de transações.

Dê preferência a colocar os comandos de controle de transações ( COMMIT, ROLLBACK , SAVEPOINT ) no ambiente chamador. Caso isto não seja possível use dentro da Stored Procedure, de preferência, apenas comandos SAVEPOINT e ROLLBACK TO *savepoint*, de modo que não interfira na transação iniciada no chamador.

Um modo de lidar com esse problema é fazer a *stored\_procedure* sinalizar uma *exception* para o ambiente chamador. Isso pode ser feito facilmente usando o comando:

```
raise_application_error ( código, mensagem )
```

Onde o código do erro deve ser um número entre -20000 e -20999 e a mensagem deverá ser um texto claro e explicativo da situação do erro.

**Importante:** Caso o ambiente chamador seja um Database Trigger, uma transação distribuída ou o SQL\*FORMS, as procedures **NÃO** podem conter nenhum comando de controle de transação.

## Transações

---

Exemplo de Controle de Transação no chamador:

```
CREATE OR REPLACE PROCEDURE
Atualiza ( cod In Number,
          aumento In Number,
          status OUT Number )
AS BEGIN
    Update tab Set val = val * aumento
    Where código = cod;
    If SQL%NOTFOUND Then
        P_processamento ( 'AVISO',
                          Tentativa de alteração - codigo' ||
                          to_char (cod)||' inexistente ');
        status := 1;
    Else
        status := 0;
    End If;
END Atualiza;

DECLARE
    status Number
BEGIN
    { Instruções }
    atualiza (ID,10,status)
    If status <> 0 Then
        Rollback;
    End If;
    {Instruções }
    Commit;
END;
```

## Saída de Dados

---

Não existem comandos específicos de entrada e saída de dados em blocos PL/SQL.

No entanto, foram criadas procedures que suportam saída de informações, permitindo retornar para o vídeo os resultados de selects ou valores de variáveis. Essas procedures enviam informações para um buffer acessível ao SQLPLUS ou SQLDBA que por sua vez podem enviá-las para tela.

Para ativar a leitura e impressão desse buffer nesses produtos digite o seguinte comando:

### **SET SERVEROUTPUT ON**

Utilizando essas procedures torna-se possível executar certas tarefas que em outro caso necessitariam do uso de uma linguagem HOST ( Ex: C, Cobol etc), além disso podemos utilizar o processo de OUTPUT como um Debug rudimentar.

Essas procedures são:

<i>Nome</i>	<i>Descrição</i>
DBMS_OUTPUT ( texto);	Insero o texto para o buffer de saída
DBMS_OUTPUT.NEW_LINE	Coloca uma marca de fim de linha no buffer de saída.
DBMS_OUTPUT.PUT_LINE	Insero o texto e coloca a marca de fim de linha no buffer.

## Saída de Dados

---

### Exemplo:

```
SQL> CREATE OR REPLACE PROCEDURE
  debuge ( taxa Number);
  AS
    aux Number;
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ( 'Começo de Execução');
    Select Sum ( salary) Into aux From emp;
    aux: = aux *taxa;
    DBMS_OUTPUT.PUT_LINE ( Valor de aux =>||
      To_char (aux));
    DBMS_OUTPUT.PUT_LINE( 'Fim de Execução');
  EXCEPTION
    When Others Then
      DBMS_OUTPUT.PUT_LINE( 'Erro na Execução');
  END;
SQL> /
SQL> SET SERVEROUTPUT ON
SQL> EXECUTE Debugando (1.30);
```

## Eliminando Procedures e Functions

---

```
>> -----DROP PROCEDURE-----nome-----
->
```

```
>> -----DROP FUNCTION -----nome-----
->
```

### Exemplo:

```
DROP PROCEDURE S_processamento;
```

```
DROP FUNCTION aumento;
```

## Administrando os Objetos - I

---

Ao criarmos qualquer objeto no banco de dados, o Oracle armazena-os em tabelas com views associadas. Essas views são divididas em três categorias ( All, Dbá e User ) e através delas obtemos informações deles. Através da view USER\_SOURCE é possível recuperar o código do objeto criado:

USER\_SOURCE ( contém o código do objeto criado ):

Name	Nome do objeto
Type	Tipo de objeto: Function , Procedure, Package, e Package Body
Line	Sequencia do Código
Text	Texto do Código

### Exemplo:

Para recuperar o texto original da Processamento crie o script source.sql com o seguinte conteúdo:

```
set pagesize 500
set linesize 80
set feedback off
set heading off
column text format a80 trunc
Select      Text
From        User_Source
Where Name = 'PROCESSAMENTO'
Order By Line;
```

```
SQL> Spool Proc.sql
SQL> @Source
```

```
PROCEDURE
processamanto ( vtipo In Varchar2, vtexto In Varchar2 )
AS
BEGIN
Insert into audite Values ( vtipo, User, Sysdate, vtexto )
EXCEPTION
When Others Then
Rollback;
End processamanto
```

```
SQL> Spool Off
```

Obs: Apesar de podermos recuperar o código de uma procedure, function ou package através do dicionário de dados, é aconselhável criarmos um arquivo com seus respectivos códigos.

## Administrando os Objetos - I

---

Através da View `USER_ERRORS` é possível recuperar que tipo de erro ocorreu durante a criação do objeto: `USER_ERRORS` ( contém os erros da última compilação ):

Name	Nome do Objeto
Type	Tipo do Objeto: Function , Procedure, Package, e Package Body
Sequence	Sequencia de Erros
Line	Linhas do código com Erro
Position	Posição do erro na Linha do Código
Text	Texto do Erro

### Exemplo:

Caso tentemos criar a procedure processamento com o script:

```
CREATE OR REPLACE PROCEDURE
processamento ( vtipo In Varchar2, vtexto In Varchar2)
AS
BEGIN
  Insert into auditori Values ( vtipo, User, Sysdate, vtexto)
EXCEPTION
  When Others Then
    Rollback;
End processamento;
```

Obteremos uma mensagem como:

```
Message 225 not found; product = PLUS31; facility = SP1
```

Porém se usarmos os comandos a seguir obteremos uma informação bem melhor sobre os erros:

```
set linesize 160
column POS
column DESCRIÇÃO format at 150 TRUNC
set space 2
select line || ' / ' || position POS, text DESCRIÇÃO
from user_errors;
```

POS	DESCRIÇÃO
-----	-----
5/17	PLS-00201: identifier 'AUDITORI' must be declared
5/5	PL/SQL Statement ignored

No exemplo acima o nome correto do Objeto é `AUDITE` e não `AUDITORI` **Obs:** O comando **SHOW ERROR** tem o mesmo efeito.

## Compilando

---

Ao **criarmos** uma Procedure ou Function é feita uma **compilação** desses objetos. Caso seja identificado, no momento execução, que um determinado objeto referenciado sofreu alguma **modificação**, o programa será **recompilado** e então executado.

Através de seu dicionário de dados o Oracle verifica as relações existentes entre seus objetos. Dessa forma ele verifica os status desses objetos e submete a execução ou recompila e executa.

No entanto, quando o objeto referenciado for uma **procedure ou function** que esteja em um banco de dados remoto, teremos um tratamento ligeiramente diferente após uma alteração nesses objetos. A **primeira vez** que tentarmos executar a procedure local obteremos uma mensagem de **erro**, mas na **segunda tentativa a recompilação será automática**.

No caso de uma referência a uma **tabela** ou outro objeto remoto que não uma function ou procedure **NÃO** haverá recompilação automática e será necessário submeter as procedures ou functions á uma compilação manual.

**Importante:** Um objeto pode ser referenciado direta ou indiretamente. Um objeto é referenciado diretamente quando uma procedure o cita em seu corpo e indiretamente quando a procedure se utiliza de um terceiro objeto que o manipula. Mesmo os objetos que acessam outros indiretamente necessitariam de uma nova recompilação caso houvesse alteração em qualquer nível.

```
ALTER PROCEDURE nome COMPILE
```

```
ALTER FUNCTION nome COMPILE
```

### Exemplo:

```
SQL> Alter Procedure processamento Compile
```

```
SQL> /
```

```
Procedure Altered
```

## Administrando as Referências

---

Através da View USER\_DEPENDENCIES é possível verificar quais são os objetos referenciados:

USER\_DEPENDENCIES ( contém as referências do objeto):

Name	Nome do Objeto
Type	Tipo do Objeto: Function, Procedure, Package e Package Body.
Referenced_Owner	Dono do Objeto Referenciado
Referenced_Name	Nome do Objeto Referenciado
Referenced_Type	Tipo do Objeto Referenciado

### Exemplo:

1.) Verificando os objetos diretamente referenciados:

```
SQL>      Select  Referenced_Owner ESQUEMA,
           Referenced_Name OBJETO,
           Referenced_Type TIPO
2
3         From  User_Dependencies
4         Where Name = 'PROCESSAMENTO';
```

2) Verificando os objetos indiretamente referenciados. Podemos criar um script chamado List\_Ref.sql com:

```
Drop Table Referencia;
CREATE Table Referencia AS Select * From User_Dependencies;

column Objeto format a25
column Referenciado format a25
column Tipo_Referenciado format a20

Select Lpad (' ', 2 * Level) || Name Objeto,
       Referenced_Name Referenciado,
       Referenced_Type Tipo_Referenciado
From Referencia
Where Referenced_Owner not in ( 'SYS', 'PUBLIC', 'SYSTEM')
Connect by name = prior Referenced_name
Start With NAME = 'PROCESSAMENTO';
```

Execute o script...

## Administrando os Objetos - II

---

Através da View USER\_OBJECTS conseguimos identificar qual o status de um determinado objeto após um comando de DDL:

USER\_OBJECTS( contém o status do comando de DDL):

Object_Name	Nome do Objeto
Object_Id	Identificador do Objeto
Object_Type	Tipo do Objeto
Created	Data de Criação do Objeto
Last_Ddl_Time	Data do último Comando de DDL
Time_Stamp	Status do Objeto: VALID, INVALID

### Exemplo:

Podemos criar o script OBJ.sql com:

```
Column Objeto Format a20
Column Tipo Format a20
Column Status Format a10
```

```
Select Object_Name Objeto, Object_Type, Status
From user_objects
Where Object_Name = 'PROCESSAMENTO';
```

SQL> @Obj

OBJETO	OBJECT_TYPE	STATUS
-----	-----	-----
PROCESSAMENTO	PROCEDURE	VALID

```
SQL> Alter Table AUDITE Modify ( Usuário Varchar2 (30));
Table altered
```

SQL> @Obj

OBJETO	OBJECT_TYPE	STATUS
-----	-----	-----
PROCESSAMENTO	PROCEDURE	INVALID

```
SQL> Alter Procedure PROCESSAMANTO Compile;
Procedure altered.
```

SQL> @Obj

OBJETO	OBJECT_TYPE	STATUS
-----	-----	-----
PROCESSAMENTO	PROCEDURE	VALID

---

## Sinônimos

---

Para fazermos referências a procedures, functions e packages de forma transparente convém criarmos **Sinônimos**.

### Exemplo:

```
SQL> Connect usuário/senha
```

```
SQL> Execute processamento ( 'Aviso', 'Exemplo de Sinônimo');end;
```

```
ERROR at line1:
```

```
ORA-06550: line1, column7:
```

```
PLS-00313: 'PROCESSAMENTO' not declared in this scope
```

```
ORA-06550: line1, column7:
```

```
PL/SQL: Statement ignored
```

```
SQL>      CREATE Synonym processamento      For SCHEMA.processamento;
```

```
      Synonym created.
```

```
SQL> Execute processamento( 'Aviso', 'Exemplo de Sinônimo');
```

```
      PL/SQL procedure successfully completed.
```

---

## Benefícios de Procedures e Functions

---

**Controle do acesso aos objetos do Banco de Dados** para usuários sem privilégio sobre eles. O usuário não necessita possuir, por exemplo o privilégio de insert em uma tabela caso exista alguma procedure que insira registros nela.

**Controle da lógica das transações do negócio** evitando que erros de programação afetem a integridade e a segurança do banco.

**Controle de código em um único lugar.**

**Evitar** o trabalho do gerenciador para compilar blocos de comandos pois estes já estão compilados.

**Reaproveitamento do código em memória** evitando parses de um mesmo comando ( ou bloco de ) chamado por vários usuários.

**Redução** no número de **chamadas** do gerenciador, reduzindo tráfego na rede.

## 5. Packages

### Entendendo Packages

---

Package é um objeto do banco de dados que agrupa procedures, funções, variáveis, exceptions e constantes que se relacionam, proporcionando uma melhor organização e visão dos aplicativos que irão utilizar-se desses objetos.

Um **PACKAGE** está dividido em duas partes:

-PACKAGE SPECIFICATION ( Definições )

-PACKAGE BODY ( Corpo )

### Entendendo Packages

---

**DECLARAÇÃO** ( Package Specification ):

Contém somente as declarações de procedures, funções, variáveis, constantes e exceptions que serão referenciadas posteriormente pelos aplicativos afins. Essas declarações são públicas. A parte declarativa é obrigatória, e através dela é feita a interface com os aplicativos. Um programador que utilize procedimentos do package precisa unicamente das informações contidas nesta parte; nenhum dado a respeito de como as rotinas são implementadas é necessário. Ele recebe informações como os nomes das variáveis declaradas no package ou quais os parâmetros que em uma determinada função recebe.

**CORPO** ( Package Body ):

Contém a definição ( código) das procedures, funções, cursors e exceptions declaradas na parte anterior ( declarações públicas ) e qualquer outro objeto necessário para o desenvolvimento do(s) subprograma(s) ( declarações privadas ). Para o desenvolvimento de sistemas esta parte não é obrigatória, ou seja durante o projeto caso não se tenha certeza de como será o fluxo de um programa mas tem-se a idéia de seu objetivo, podemos referenciá-lo apesar do programa não estar completo. Dessa forma é possível criar outros objetos ( procedures, functions, blocos PL/SQL, etc) que utilizam informações que serão codificadas posteriormente.

## Criando Packages

---

CREATE ( OR REPLACE ) PACKAGE *nome* IS (AS)

declarações de procedures

declarações de funções

declarações de variáveis

declarações de cursores

declarações de exceptions

END *nome*

CREATE ( OR REPLACE ) PACKAGE BODY *nome* IS (AS)

procedures

funções

variáveis

cursores

exceptions

END *nome*

## Criando Packages

---

**Exemplo1** ( somente a parte declarativa ) :

```
CREATE OR REPLACE PACKAGE MANTEM
AS
    Usuário Varchar2 (20): = USER;
    Procedure processamento ( Tipo In Varchar2,
                             Texto In audite.Texto%TYPE);
END MANTEM;
```

**Exemplo2** ( package somente com informações declarativas ):

```
CREATE OR REPLACE PACKAGE empresa
AS
```

```
    Usuário Varchar(20): = USER;
```

```
    Type Recemp Is RECORD (
    Código emp.id%TYPE,
    Nome emp.first_name.%TYPE);
    Regemp          Recemp;
```

```
    Type Recdept Is Record (
    Código dept.id%TYPE,
    Nome dept.name%TYPE,
    regiao dept.regiao_id%TYPE);
    Regdept          Recdept;
```

```
END empresa;
```

## Criando Packages

---

**Exemplo3** ( Package completa):

```
CREATE OR REPLACE PACKAGE MANTEM
AS
    Usuário Varchar2(20): = USER;
    Procedure processamento( Tipo In Varchar2,
    Texto In audite.Texto%TYPE);

END MANTEM;

CREATE OR REPLACE PACKAGE BODY MANTEM
IS
PROCEDURE processamento ( vtipo In Varchar2, vtexto In Varchar2)
    AS
    BEGIN
        Savepoint Safe_Add;
        Insert into audite Values (vtipo, Usuário,
                                Sysdate, vtexto);
    EXCEPTION
        When Others Then
            Rollback to Safe_Add
    End Processamento;

END MANTEM;
```

**OBS:** Podemos criar separadamente cada parte de um package.

## Referenciando objetos da Package

---

### No SQL/PLUS:

```
SQL> EXECUTE MANTEM.processamento('AVISO', 'Package');
```

### No PL/SQL

```
DECLARE
    { declaração de variáveis}
BEGIN
    {Instruções}
    Select * Into regemp
    From emp;
    { Instruções}
    MANTEM.processamento ( 'AVISO', 'empregados OK');
END;
```

### No SQLFORMS ( Dentro de um trigger ou Procedure)

```
If : BLOCO1.CAMPO1 = 'S' Then
    MANTEM.processamento('AVISO', 'empregados OK ');
Else
    ROLLBACK;
End If;
```

## Benefícios da Package

---

Packages proporcionam um conjunto de benefícios tanto durante o desenvolvimento de sistemas como em sua manutenção e execução.

### DURANTE O DESENVOLVIMENTO:

#### **Modularidade:**

Compreendendo os relacionamentos lógicos existentes em um determinado sistema, podemos encapsulá-los através de um package. Dessa forma cada package é de fácil compreensão, as interfaces são simples, claras e bem definidas.

#### **Independência de Compilação:**

Para compilar um programa que use o package basta que a especificação do package esteja criada.

#### **Controle do Código e Objetos:**

Especificando quais objetos são públicos ( visíveis e acessíveis) e quais são privados ( definidos e utilizados internamente ).

## Benefícios da Package

---

### **DURANTE A MANUTENÇÃO:**

#### **Fácil manutenção:**

Ao se alterar o corpo de um package, diferentemente das procedures e functions , não é necessário a recompilação dos objetos que fazem referência a parte do package body alterada. Mas uma alteração no package specification necessitará de uma recompilação.

### **DURANTE A EXECUÇÃO:**

#### **Performance:**

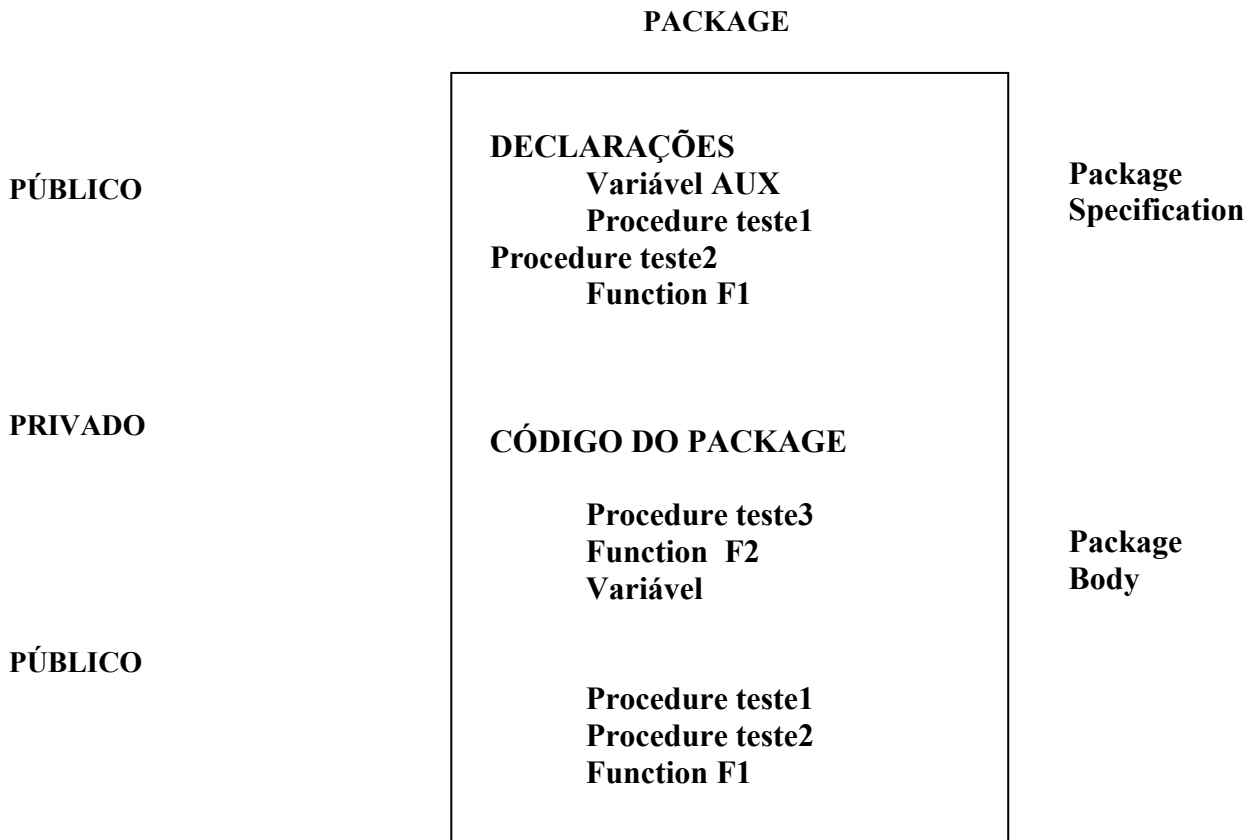
Quando um programa definido em um package é chamado, todo o package é carregado para a memória, necessitando de menos I/O nos próximos acessos.

#### **Controle em uma sessão:**

Cada sessão aberta tem seu próprio set de constantes, cursores e variáveis. Estas possuem valores null na primeira vez que um package é referenciado. Ao término de uma sessão os valores são perdidos e 'startados' com valores null numa nova sessão aberta.

**Escopo da Package**

---



Qualquer objeto declarado no **PACKAGE SPECIFICATION** será **PÚBLICO** para qualquer ambiente que o **referencie**.

Qualquer objeto declarado **SOMENTE no PACKAGE BODY** será **PRIVADO**, **NÃO** podendo ser **referenciado** por outros ambientes. Porém, internamente, esses objetos são públicos.

## Procedures Multi-Datatype

---

Em certas situações é interessante que uma determinada função possa receber mais de um tipo de dado. Neste caso criamos no package várias funções com o mesmo nome, cada uma aceitando um tipo de dado diferente. No momento da chamada o Oracle determinará qual função deverá ser acionada.

### Exemplo:

#### Programa PL/SQL (Teste.sql):

```
SET SERVEROUTPUT ON
BEGIN
/* Programa chama a mesma procedure passando tipos diferentes
*/
    Multi_Type.Teste ('Tipo CHARACTER', 'W');
    Multi_Type.Teste ( 'Tipo NUMERICO', 4);
    Multi_Type.Teste ( 'Tipo DATA', SYSDATE);

END;
```

#### Resultado da Execução:

```
SQL> @Teste
CHAR=> P1 = Tipo CHARACTER      P2=W
NUM=>  P1 = Tipo NUMERICO       P2=4
DATE=> P1 = Tipo DATA          P2=14/06/94
PL/SQL procedure successfully completed.
```

## Procedures Multi-Datatype

---

### Packages Specification:

```
CREATE OR REPLACE PACKAGE Multi_Type
is
    PROCEDURE Teste ( Y IN CHAR, Z IN NUMBER);
    PROCEDURE Teste ( Y IN CHAR, Z IN CHAR);
    PROCEDURE Teste ( Y IN CHAR, Z IN DATE);

END Multi_Type;
/
```

### Package Body

```
CREATE OR REPLACE PACKAGE BODY Multi_Type
is
    PROCEDURE Teste ( Y IN CHAR, Z IN NUMBER) is
    BEGIN
        DBMS_OUTPUT.PUT_LINE ('NUM => '||
            'P1 = '|| Y||
            'P2 = '|| TO_CHAR(Z));
    END;

    PROCEDURE Teste ( Y IN CHAR, Z IN CHAR ) is
    BEGIN
        DBMS_OUTPUT.PUT_LINE ( 'CHAR =>'||
            'P1 = '|| Y||
            'P2 = '|| Z);
    END;

    PROCEDURE Teste ( Y IN CHAR, Z IN DATE ) is
    BEGIN
        DBMS_OUTPUT.PUT_LINE(' DATE => '||
            'P1 = '|| Y||
            'P2 = '|| TO_CHAR(Z, ' DD/MM/YY'));
    END;

END Multi_Type;
```

---

## Eliminando Packages

---

### Exemplo:

DROP PACKAGE mantem;

DROP PACKAGE BODY mantem;

**Obs:** O comando DROP PACKAGE elimina tanto a especificação quanto o corpo do Package.

---

## Compilando

---

Para compilar um Package manualmente execute os seguintes comandos:

**Alter Package** nome do package **Compile**

Esse comando recompilará todo o package.

**Alter Package** nome do package **Compile Specification**

Esse comando recompilará o Package Specification

**Alter Package** nome do package **Compile Body**

Esse comando recompilará o Package Body.

---

## Exercícios

---

1) Crie um package EMP\_NOME com uma função que retorne os nomes do empregado associado ao ID passado como parâmetro.

Exemplo: EMP\_NOME(20) => 'Chad'.

2) Crie um package que manipule a tabela AUDITE, implemente como a seguir funções:

- a) Usr\_Log.Inclue(tipo,texto) => Semelhante ao processamento.
- b) Usr\_Log.Lista1 (tipo) => Lista todas as entradas como tipo especificado.
- c) Usr\_Log.Lista2 (date) => Lista todas as datas a partir da data especificada.
- d) Usr\_Log.Elimina (date) => Apaga todas as entradas mais velhas que a data e hora especificadas.

## 6. Database Triggers

### Criando Database Triggers

---

**Exemplo:**

```
CREATE OR REPLACE TRIGGER Log_Alteração
BEFORE DELETE OR UPDATE ON emp
FOR EACH ROW
WHEN ( : new.salary > 2500)
BEGIN
    processamento ('ALTERAÇÃO', user, sysdate, 'alterado salarial');
EXCEPTION
    When Others
        Raise_application_error ( - 20000, ' Trigger Failure');
END;
```

## Ativação de Triggers

---

Triggers são tipos especiais de stored procedures que são ativados automaticamente como efeito de um comando de DML em uma tabela. Logo:

**Não** é possível chamar explicitamente um trigger.

**Não** é possível passar parâmetros para um triggers.

**Não** é possível associar um trigger a qualquer objeto que não seja uma tabela ( ex: views, synonyms etc..)

Um trigger pode ser ativado por qualquer um dos comandos de DML.

Exemplo:

```
CREATE TRIGGER tr_ins AFTER INSERT ON tab
CREATE TRIGGER tr_del AFTER DELETE ON tab
CREATE TRIGGER tr_upd AFTER UPDATE ON tab
```

Ou por qualquer conjunto de eventos.

**Exemplo:**

```
CREATE TRIGGER tr_all
AFTER INSERT OR DELETE OR UPDATE ON tab
```

Além disso se o trigger especificar um UPDATE como evento é possível incluir uma lista de colunas opcional no comando.

**Exemplo:**

```
CREATE TRIGGER tr_col
AFTER INSERT OR UPDATE OF codsocio, nomsocio ON tab.
```

## Cláusulas BEFORE/AFTER

---

Quanto ao momento de execução os triggers podem ser disparados:

**BEFORE evento** : O trigger é executado após a submissão do comando mas antes da sua execução. Os dados são lidos e passados para o trigger e caso o trigger seja bem sucedido eles são lidos novamente para a execução da operação. Caso o trigger falhe, a operação é suspensa sem alterar o banco.

Sugestão: trigger\_up\_date

**AFTER evento**: O trigger é executado após a execução do comando mas antes da sua liberação para o usuário. Os dados são lidos para a execução da operação e passados para o trigger. Assim a performance é um pouco **melhor** do que na opção BEFORE. Caso o trigger falhe, a operação é desfeita no banco.

## Triggers de Linha ou de Comando

---

Os triggers podem ser ativados uma única vez por comando ou a cada linha manipulada. O default é a ativação por comando sendo a ativação por linha definida pela cláusula **FOR EACHROW**.

### Exemplo:

Na tabela:

	<b>Cod_socio</b>	<b>Nom_socio</b>
1	Pedro	
	1Eliane	
	2Celso	
	3Cynthia	
	4Eduardo	

O comando: `DELETE tab WHERE cod_socio > 2;`

Ativaria apenas **uma** vez o trigger:

```
CREATE TRIGGER tr_com
BEFORE DELETE ON tab
BEGIN
    { comando}
END;
```

Mas ativaria **três** vezes o trigger:

```
CREATE TRIGGER tr_com
BEFORE DELETE ON tab
FOR EACH ROW
BEGIN
    { comando}
END;
```

## Sequência de Acionamento

---

Um evento pode ser associado a até 4 triggers que serão executados na seguinte ordem:

- BEFORE** por comando
- BEFORE** por linha
- AFTER** por linha
- AFTER** por comando

### Exemplo:

O comando: DELETE Locação WHERE cod\_socio = 1;

Poderia acionar:

**BEFORE por comando ->**

BEFORE por linha -> ( para cada linha )

AFTER por linha -> ( Para cada linha )

**AFTER por comando ->**

### Acessando Valores de Colunas

---

É possível pesquisar dentro de um trigger de linha, os valores das colunas antes e depois da execução do comando ativador através de duas variáveis de registro.

**: old** -> contém os valores antes da alteração  
**: new** -> contém os valores depois da alteração

Seguindo a seguinte tabela:

<b>Evento</b>	<b>:new</b>	<b>:old</b>
UPDATE	nova linha	linha velha
DELETE	NULL	linha velha
INSERT	nova linha	NULL

#### Exemplo:

```
CREATE TRIGGER tr_expl
BEFORE UPDATE ON emp
FOR EACH ROW
BEGIN
    If (:new.salary / : old.salary ) > 1.80 Then
        processamento('ERRO', 'Aumento maior que 80%');
    END If;
END;
```

### Alterando Valores de Colunas

---

É possível os valores de qualquer coluna, de todas as linhas afetadas pelo evento ativador, dentro de um trigger do tipo BEFORE. Para isso basta alterar os valores ligados a variável : new.

#### Exemplo:

Evento chamador: UPDATE empregados  
 SET salario = 1000;

Trigger:

```
CREATE TRIGGER Acerta_Salario
BEFORE UPDATE ON emp
FOR EACH ROW
BEGIN
    /* Garante que não haverá aumentos de mais que 80% */
    : new.salary := MIN ( : new.salary, : old.salary * 1.8);
    If :new.manager_id is null Then
        /* Calcula a comissão da Presidencia */
        : new.commission_pct := :new.salary * :old.commission_pct;
    END If;
END;
```

---

## Cláusula WHEN

---

Opcionalmente, uma restrição pode ser incluída na definição do trigger especificando a cláusula WHEN com uma expressão booleana. Essa expressão é avaliada cada vez que o trigger é acionado. Caso ela receba o valor TRUE o corpo do trigger é executado normalmente, entretanto caso o valor da expressão seja FALSE ou NULL estão o trigger não executado.

Quando a cláusula WHEN é especificada é obrigatório a inclusão da cláusula FOR EACH ROW. Isto é, a restrição do WHEN só é válida para triggers de linha e é avaliada individualmente para cada linha afetada pelo comando disparador do trigger.

### Exemplo:

```
CREATE TRIGGER tr_expl
BEFORE UPDATE ON emp
FOR EACH ROW
WHEN ( ( :new.salary / :old.salary ) > 1.80 )
BEGIN
    processamento ('ERRO', 'Aumento maior que 80%');
END;
```

---

## Descobrimo o Evento Chamador

---

Se um trigger trata mais de um tipo de evento, é possível descobrir qual foi o evento chamador testando certas condições especiais:

- INSERTING**
- DELETING**
- UPDATE** ou **UPDATING** ( 'nome da coluna'

### Exemplo

```
CREATE TRIGGER tr_expl
BEFORE DELETE
OR UPDATE OF cod_socio, nom_socio
ON tab
FOR EACH ROW
BEGIN
    If DELETING Then
        sp_addlog( 'AVISO', 'Deletando o socio ' || to_char(:old,cod_socio));
    END If;
    If UPDATING ( 'COD_SOCIO') Then
        sp_addlog( 'AVISO', 'Alterando o socio ' || to_char( : new.cod_socio));
    End If;
    If UPDATING ( 'NOM_SOCIO') Then
        sp_addlog( 'AVISO', 'Alterando o nome do socio'|| :old.nom_socio||'para '||:new.nom_socio);
    End If;
END;
```

## Administrando Triggers - I

---

```
>>-----ALTER TRIGGER-----nome-----DISABLE-----
>>-----ALTER TRIGGER-----nome-----ENABLE-----
>>-----DROP TRIGGER-----nome-----
>>-----ALTER TABLE -----nome-----DISABLE ALL TRIGGERS-----
>>-----ALTER TABLE-----nome-----ENABLE ALL TRIGGERS-----
```

Quando um trigger é criado ele é automaticamente habilitado. Normalmente a pequena perda de performance devido o acionamento do trigger é bem tolerada, mas existem certas atividades em que seu funcionamento não é desejável, por exemplo:

Durante um **Insert as Select** ou uma carga com **IMP** ou **SQL\*Loader** em que o usuário tenha certeza de que **nenhum dado** entrado na tabela **violará** qualquer restrição de **integridade** testado pelo trigger.

Durante uma manutenção no banco de dados em que o usuário tenha a **certeza** de que o **trigger falhará**, pois a integridade já está comprometida.

Durante qualquer evento que deixe objetos referenciados pelo trigger não disponíveis. Isso pode ocorrer devido a uma falha de rede ( no caso de objetos remotos) ou a uma tablespace colocada “off-line”.

Para desabilitar temporariamente um trigger use o comando ALTER TRIGGER com a opção DISABLE. Para reabilitá-lo use a opção ENABLE. Para destruir um trigger use o comando DROP TRIGGER.

Para desabilitar temporariamente **todos** os triggers associados a uma determinada tabela use o comando ALTER TABLE com a opção DISABLE ALL TRIGGERS. Para reabilitá-los, use a opção ENABLE ALL TRIGGERS. O comando DROP TABLE automaticamente destrói todos os triggers associados a tabela destruída.

### Exemplo:

```
ALTER TRIGGER tr_expl DISABLE;
ALTER TRIGGER tr_expl ENABLE;
DROP TRIGGER tr_expl;
ALTER TABLE locação DISABLE ALL TRIGGER;
ALTER TABLE locação ENABLE ALL TRIGGER;
```

## Triggers em Cascata

---

Um trigger ativado por uma alteração em uma tabela pode, eventualmente, fazer uma alteração em outra tabela. Se esta segunda tabela também possuir um trigger ele também será disparado. Quando isso acontece é dito que os dois triggers estão em cascata.

O número de triggers que podem ficar em cascata é limitado pela configuração do Oracle. Caso ocorram problemas, parâmetros como OPEN\_CURSORS devem ser alterados.

### Exemplo:

```
CREATE TRIGGER acerta_locação
BEFORE INSERT ON item_locação
FOR EACH ROW
BEGIN
    Update locação
    Set val_total = val_total + :new.val_locação
    Where num_locação = :new.num_locação;
END;
```

```
CREATE TRIGGER loga_locação
BEFORE UPDATE ON locação
FOR EACH ROW
BEGIN
    processamento ( 'AVISO', 'Alterando o valor total da locação: ' ||
                    to_char ( :new.num_locação);
END;
```

---

## Exercícios

1) Crie uma tabela com o seguinte comando:

```
CREATE TABLE EMPLOG
(
    USUARIO          VARCHAR2(20),
    OLDSALARY        NUMBER(10,2),
    NEWSALARY        NUMBER(10,2),
    DATAALTERACAO  DATE);
```

Crie uma trigger na tabela EMP para que qualquer alteração feita na table EMP no campo SALARY seja registrada na tabela EMPLOG, o usuário que alterou, o salário antigo e o novo e a data da alteração.

## Comandos Válidos em um Trigger

---

É possível utilizar dentro do corpo de um trigger os seguintes tipos de comando:

- Qualquer comando ou estrutura do PL/SQL 2.0
- SELECT ( desde que SELECT INTO ou cursor )
- UPDATE, DELETE ou INSERT

**Não** são permitidos os seguinte tipos de comando:

- Qualquer comando de DDL ( CREATE, DROP, etc...)
- ROLLBACK, COMMIT ou SAVEPOINT
- Qualquer procedure que contenha um dos comandos acima

Quando houver uma condição em que exista a necessidade de evitar o sucesso do comando chamador sinalize uma exception com a procedure **raise\_application\_error**:

**Exemplo:**

```
CREATE TRIGGER tr_expl
BEFORE UPDATE ON emp
FOR EACH ROW
BEGIN
    If ( :new.salary / :old.salary ) > 1.80 Then
        Insert into emplog values (user, :old.salary, :new.salary,sysdate);
        raise_application_error ( -20000, ' TriggerFailure');
    End If;
END;
```

## Outras Restrições

---

No caso de um trigger de linha existem três conceitos adicionais que devem ser analisados:

**Tabelas Mutantes:** Tabelas alteradas pelo evento chamador ou alteradas através de uma opção CASCADE de uma restrição de integridade referencial.

**Tabelas de Restrição:** Tabelas que referenciam tabelas mutantes através de uma declaração de restrição de integridade referencial.

**Tabelas Independentes:** Tabelas que não se relacionam a nível de integridade referencial a nenhuma tabela afetada pelo evento disparador.

## Outras Restrições

---

Com relação a tabelas mutantes temos a seguinte regra:

**Não** é permitido executar , no corpo do trigger de linha, qualquer comando de DML ( SELECT, INSERT, DELETE ou UPDATE) sobre as tabelas mutantes.

### Exemplo:

```
CREATE TRIGGER Errol
AFTER DELETE ON Titulo
FOR EACH ROW
DECLARE
    Contador Number;
BEGIN
    /* ERRO devido a acesso na tabela mutante Fita */
    Update Fita
        Set cod_titulo = 0
        Where cod_titulo = :old.cod_titulo;

    /* ERRO devido a acesso na tabela mutante Titulo */
    Select count (*)
        Into contador
        From Titulo;
END;
```

Com relação a tabela de restrição temos a seguinte regra:

**Não** é permitido executar, no corpo de trigger de linha, um comando que altere uma coluna declarada em um PRIMARY KEY, FOREIGN ou UNIQUE de uma tabela de restrição.

### Exemplo:

```
CREATE TRIGGER Erro2
AFTER DELETE ON Titulo
FOR EACH ROW
BEGIN
    /* ERRO devido a alteração da coluna FOREIGN KEY
    cod_titulo na tabela de restrição Fita */
    If Updating ( 'COD_TITULO') Then
        Update Fita */
        set cod_titulo = :new.cod_titulo
        where cod_titulo = :old.cod_titulo;
    End If;
END;
```

## Permissões e Triggers

---

Do ponto de vista das permissões de acesso, um trigger é executado tomando como referência o criador do trigger. Assim todos os objetos referenciados pelo trigger na sua definição deverão ser acessíveis ao usuário criador do trigger na época de seu acionamento por parte de qualquer usuário. Se o criador do trigger perder os acessos necessários para sua execução, o trigger falhará mesmo que o usuário acionador tenha esses acessos.

No exemplo a seguir temos duas situações:

- O usuário de desenvolvimento cria um trigger que acessa uma tabela de log do sistema na qual ele tem permissão de escrita. Um usuário comum pode inserir dados na tabela de fita e conseqüentemente acionar o trigger sem problemas.
- O usuário System erradamente retira os privilégios de escrita do usuário de desenvolvimento e libera esses privilégios para o usuário comum. Neste caso o trigger falha.

## Permissões e Triggers

---

```
SQL> Connect System/Manager
SQL> Grant All On EMPLOG To usuário;
SQL> Connect aluno1/aluno1;
SQL> CREATE trigger add_log Before Update On EMP
      Begin
          Insert Into SYSTEM.EMPLOG
          Values (user,.....);
      End;
SQL> Grant Update On emp To Usuário;
SQL> Connect Usuário/Usuário
SQL> Update emp Set salary = 1500 Where id = 10;
SQL> Rem O Update tem sucesso e o log e gerado
```

```
SQL> Connect System/Manager
SQL> Grant All On EMPLOG To Usuário;
SQL> Revoke All On EMPLOG From aluno1;
SQL> Connect Usuário/Usuário
SQL> Update emp Set salary = 1500 Where id = 10;
SQL> Rem O Update Falha
```

## Exemplos de Uso de Trigger

---

Na criação de logs o uso de triggers possibilita uma grande flexibilidade. No exemplo abaixo o usuário é forçado a registrar em uma variável do package AUDIT chamada MOTIVO, qual o motivo da alteração feita.

**Exemplo:**

```
CREATE TRIGGER grava_log
BEFORE DELETE ON emp
  Aux Number;
BEGIN
  If Audit.Motivo is Null Then
    Raise_Application_Error( - 20000, 'Trigger Failure');
  Else
    processamento( 'AVISO', Socio||
                  To_Char(cod_socio) || 'deletado');
    processamento('MOTIVO', Audit.Motivo);
  End If
END;
```

-----

É possível melhorar a segurança de maneira não provida pelos recursos normais do Oracle.

**Exemplo:**

```
CREATE TRIGGER improve_sec
BEFORE INSERT OR DELETE OR UPDATE ON emp
  Aux Number;
BEGIN
  Aux := To_Number( Sysdate, 'HH24')
  If aux Not Between 9 And 18 Then
    processamento( 'AVISO', Tentativa de acesso as '
                  || To_Char(aux) || 'horas')
    Raise_Application_Error(-2000, 'Trigger Failure');
  End If;
END;
```

## Exemplos de Uso de Trigger

---

É Frequentemente útil criar defaults variáveis para campos opcionais. No exemplo abaixo a fita só é incluída como ativa durante a primeira semana do mês.

### Exemplo:

```
CREATE TRIGGER seta_defaults
BEFORE INSERT ON fita
FOR EACH ROW
BEGIN
    If:new.ind_ativa is Null
    And To_Numbers(Sysdate, 'DD') <= 7 Then
        :new.ind_ativa := 'S';
    Else
        :new.ind_ativa := 'N';
    End If;

    If:new.ind_locada is Null Then
        :new.ind_locada := 'N';
    End If;
END;
```

## Administrando Triggers – II

---

Assim como no caso de Stored Procedures é possível pesquisar informações sobre um trigger criado pelo usuário no dicionário de dados. Isto é feito através de queries nas view USER\_TRIGGERS e USER\_TRIGGER\_COLS.

A User\_Triggers mostra:

<b>Trigger_Name</b>	Nome do Trigger
<b>Trigger_Type</b>	Tipo( Before/After & Comando/Linha)
<b>Triggering_Event</b>	Eventos Disparadores
(INSERT/DELETE/UPDATE)	
<b>Table_Name</b>	Dono da tabela associada ao trigger
<b>Referencing_names</b>	Nomes atuais das variáveis new e old
<b>Table_Name</b>	Nome da tabela associada ao trigger
<b>When Clause</b>	Condição especificada no When
<b>Status</b>	Indica se o trigger está habilitado ou não
<b>Description</b>	Texto com o cabeçalho do trigger
<b>Trigger_Body</b>	Texto com o corpo do trigger

### Exemplo:

Para gerar um arquivo com o comando de criação do trigger Gera\_Log:

```
SQL> Spool cria_trg
SQL> Select'CREATE TRIGGER', description, trigger_body
        From User_Triggers
        Where Trigger_Name = 'GERA_LOG';
SQL> Spool Off
```

**Obs:** Assim como nos packages e stored procedures, é aconselhável guardar os scripts SQL de criação dos triggers.

## Administrando Triggers – II

---

A User\_trigger\_Cols armazena informações sobre o uso de colunas no trigger. A User\_Trigger\_Cols mostra:

<b>Trigger_Owner</b>	Dono do Trigger(geralmente igual a Table_Owner)
<b>Table_Name</b>	Nome do Trigger
<b>Table_Owner</b>	Dono da tabela associada ao trigger
<b>Table_Name</b>	Nome da tabela associada ao trigger
<b>Column_List</b>	Indica se a coluna aparece na cláusula Update(Y/N)
<b>Column_Usage</b>	Indica o tipo de uso

### Exemplo:

```
CREATE TRIGGER Teste
BEFORE UPDATE OF ind_ativa ON Fita FOR EACH ROW
BEGIN
    If:old.ind_ativa = 'S' And :new.ind_ativa = 'N'
    And :old.ind_locada = 'S'
        :new.ind_ativa := 'P';
    End If;
END;
```

Teremos:

<b>Column_Name</b>	<b>Column_List</b>	<b>Column_Usage</b>
Ind_Ativa	YES	NEW IN OUT OLD IN
Ind_Locada	NO	OLD IN

Indicando que Ind\_Ativa foi declarada na lista do evento Update e que seu valor NEW foi usado para ler e alterar o valor do campo e seu valor OLD foi usado para leitura. A coluna Ind\_Locada tem seu valor OLD lido pelo trigger.

## Exercícios

---

1 ) Crie um conjunto de triggers que controle operações de DML nesta tabela, seguindo as seguintes regras de negócio:

- a) Devido ao excesso de demissão de funcionários, a exclusão do banco só pode ocorrer nas Sexta feiras..
- b) Todas as operações de Insert, Delete ou Update na tabela de EMP devem ser registradas em uma tabela de log.

## Variáveis do Tipo Cursor

---

A versão 2.3 do PL/SQL, disponível a partir do Oracle Server Versão 7.3, permite que se crie e utilize variáveis do tipo cursor. Da mesma forma que o cursor convencional, a variável do tipo cursor aponta para a linha corrente no conjunto de resultados de uma consulta que retorne múltiplas linhas, mas ao contrário do cursor estático, que está vinculado à uma única consulta, as variáveis cursor podem ser associadas a várias consultas, até mesmo dentro de um mesmo programa. Variáveis do tipo cursor são verdadeiras variáveis PL/SQL, você pode associar novos valores a ela e passa-la como argumento a subprogramas.

Uma variável do tipo cursor é como um ponteiro nas linguagens C e Pascal, ela guarda a posição na memória (endereço) de um objeto ao invés de guardar o próprio objeto. Portanto quando você declara uma variável do tipo cursor você está criando um ponteiro e não um objeto.

O principal benefício da variável do tipo cursor é que ela provê um mecanismo para se passar resultados de consultas entre diferentes programas PL/SQL, ou ainda entre programa PL/SQL cliente e programa servidor. Em um ambiente cliente/servidor, por exemplo, um programa do lado cliente poderia abrir a variável do tipo cursor e começar a extrair seus dados, e então passar essa variável como argumento a uma procedure armazenada no servidor. Este programa poderia então continuar a extrair seus dados e passar a variável de volta ao cliente para que este feche o cursor.

### Características das variáveis cursor

As variáveis do tipo cursor permitem que você:

Associe uma variável do tipo cursor com diferentes consultas em tempos diferentes durante a execução de seu programa. Uma variável do tipo cursor pode ser usada para extrair dados de diferentes conjuntos de resultados.

Passe uma variável do tipo cursor como argumento a um procedimento ou função. Você pode compartilhar o resultado de uma consulta com outros procedimentos.

Empregue toda a funcionalidade de cursores estáticos. Você pode utilizar as declarações OPEN, FETCH e CLOSE e referenciar os atributos %ISOPEN, %FOUND, %NOTFOUND e %ROWCOUNT em variáveis do tipo cursor.

Associar o conteúdo de uma variável do tipo cursor (inclusive seu conjunto de resultados) a outra variável do tipo cursor.

### Declarando o tipo REF CURSOR e a variável do tipo cursor

A criação de variáveis do tipo cursor é feita em duas etapas: primeiro você define um tipo REF CURSOR e então declara a variável como sendo daquele tipo.

A sintaxe para se criar um tipo de referência a cursor é a seguinte:

```
TYPE nome_tipo_cursor IS REF CURSOR [RETURN tipo_retornado];
```

Onde nome\_tipo\_cursor é o nome do tipo e tipo\_retornado é a especificação do dado retornado pelo tipo cursor. O tipo\_retornado pode ser qualquer estrutura válida para uma cláusula RETURN de um cursor normal, definida usando o atributo %ROWTYPE ou referenciando um registro (record) previamente definido.

A cláusula RETURN é opcional, e quando usada, o tipo é dito "forte" pois fica atado a um tipo record, ou tipo row. Qualquer variável do tipo cursor declarada de um tipo "forte" pode apenas ser utilizada com declarações SQL que retornem dados do mesmo tipo da declaração usada na cláusula RETURN.

Por outro lado, o tipo cursor que não possui a cláusula RETURN é dito "fraco" e pode ser utilizado de formas muito mais flexíveis, isto é, pode ser utilizado com consultas que retornem qualquer estrutura de dados.

Uma vez declarado o tipo REF CURSOR você pode declarar a variável daquele tipo, como mostrado no seguinte exemplo:

```
DECLARE
    -- Criando o tipo
    TYPE DeptCurTyp IS REF CURSOR RETURN dept%ROWTYPE;

    -- Criando a variável
    dept_cv DeptCurTyp;
BEGIN
    ...
END;
```

É importante lembrar que a declaração de uma variável do tipo cursor não cria um objeto cursor. Para que isto ocorra é necessário usar a sintaxe OPEN FOR, que cria um novo objeto cursor e o associa à variável.

Outra observação importante é que variáveis cursor não possuem persistência de estado por serem apenas ponteiros, não podendo portanto serem referenciadas após o fim da execução do procedimento que abriu o cursor.

### Abrindo uma variável do tipo cursor (OPEN - FOR)

A declaração OPEN - FOR associa o cursor com uma consulta que retorne múltiplas linhas, executa a consulta e identifica o conjunto de respostas. A sintaxe para a declaração é:

```
OPEN {nome_variável_cursor | :variável_cursor_de_ambiente} FOR declaração_sql;
onde variável_cursor_de_ambiente é uma variável declarada em um ambiente PL/SQL como o SQL*Plus ou programa Pro*C, e declaração_sql é qualquer declaração SELECT que não possua a cláusula FOR UPDATE.
```

Outras declarações OPEN - FOR podem abrir a mesma variável do tipo cursor para diferentes consultas, não sendo necessário para isto fecha-lo antes.

Extraindo dados da variável do tipo cursor (FETCH)

Assim como com cursores estáticos, para se obter o resultado da consulta é utilizada a declaração FETCH que extrai as linhas uma a uma da variável do tipo cursor, e possui a seguinte sintaxe:

```
FETCH {nome_variável_cursor | :variável_cursor_de_ambiente} INTO registro;
FETCH {nome_variável_cursor | :variável_cursor_de_ambiente} INTO variável1 [,variável2 ...];
```

Quando a variável do tipo cursor foi declarada como sendo de um tipo "forte", o compilador PL/SQL verifica se a estrutura dos dados após a cláusula INTO são compatíveis com a estrutura da consulta associada à variável do tipo cursor, verifica também se o número de variáveis é correspondente ao número de colunas retornadas pela consulta. Caso contrário será gerado um erro.

O erro irá ocorrer em tempo de compilação se a variável for de um tipo "forte" e em tempo de execução se a variável for de um tipo "fraco". Em tempo de execução, a PL/SQL evoca a exception ROWTYPE\_MISMATCH.

#### Fechando uma variável do tipo cursor (CLOSE)

A declaração CLOSE desabilita a variável do tipo cursor. Depois disto o conjunto de resultados associado é indefinido. A sintaxe é a seguinte:

```
CLOSE {nome_variável_cursor | :variável_cursor_de_ambiente};
```

#### Exemplos de variáveis cursor

##### Exemplo 1

Este exemplo cria uma package chamada LOJA com a procedure PRODUTO que retorna os dados referentes ao produto passado como argumento. Esta procedure utiliza uma variável do tipo cursor para fazer a consulta dependendo do tipo do produto, se o produto for um livro (código 1) o cursor será associado a uma consulta à tabela LIVROS, caso o produto seja um disco (código 2) o cursor será associado a uma consulta à tabela DISCOS.

```
-- =====
--          PACKAGE ESPECIFICATION
-- =====
CREATE OR REPLACE PACKAGE LOJA IS

    PROCEDURE PRODUTO(título_in IN VARCHAR2);

END LOJA;
/

-- =====
--          PACKAGE BODY
-- =====
CREATE OR REPLACE PACKAGE BODY LOJA IS

FUNCTION QUAL_CODIGO(título_in IN VARCHAR2) RETURN NUMBER IS
    código_produto NUMBER;
BEGIN
```

```

SELECT codigo INTO código_produto
FROM TITULOS
WHERE TITULO = título_in;

RETURN código_produto;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RETURN NULL;
END QUAL_CODIGO;

PROCEDURE PRODUTO(título_in IN VARCHAR2) IS
TYPE tipo_cursor IS REF CURSOR; -- Tipo cursor (um tipo "fraco")
var_cursor tipo_cursor; -- A variável cursor
código_produto NUMBER;
livros_rec LIVROS%ROWTYPE; -- Um record
discos_rec DISCOS%ROWTYPE; -- Um record
BEGIN
código_produto := QUAL_CODIGO(título_in);

IF código_produto IS NULL THEN
DBMS_OUTPUT.PUT_LINE('Produto não cadastrado.');
```

```

ELSE
IF código_produto = 1 THEN
OPEN var_cursor FOR SELECT * FROM LIVROS WHERE TITULO = título_in;
FETCH var_cursor INTO livros_rec;
DBMS_OUTPUT.PUT_LINE('TÍTULO : ||livros_rec.titulo);
DBMS_OUTPUT.PUT_LINE('EDITORA : ||livros_rec.editora);
DBMS_OUTPUT.PUT_LINE('PREÇO : ||livros_rec.preço);
ELSE
OPEN var_cursor FOR SELECT * FROM DISCOS WHERE TITULO = título_in;
FETCH var_cursor INTO discos_rec;
DBMS_OUTPUT.PUT_LINE('TÍTULO : ||discos_rec.titulo);
DBMS_OUTPUT.PUT_LINE('PREÇO : ||discos_rec.preço);
END IF;
CLOSE var_cursor;
END IF;
END PRODUTO;

END LOJA;
/
```

A package pode ser criada, supondo-se que o código esteja no arquivo LOJA.SQL e que o ambiente utilizando seja o SQL\*Plus, da seguinte forma:

```
SQL> @LOJA.SQL
```

E sua utilização é da forma:

```
SQL> EXECUTE LOJA.PRODUTO('ORACLE PL/SQL PROGRAMMING');
```

que retornaria:

```
TÍTULO : ORACLE PL/SQL PROGRAMMING
EDITORA : O'REILLY & ASSOCIATES, INC.
```

PREÇO : 12

PL/SQL procedure successfully completed.

### Exemplo 2

Uma outra forma de utilização de variáveis do tipo cursor é mostrada neste segundo exemplo. É criada uma package com declarações de tipos para variáveis cursor e procedimentos para abrir o cursor e para extrair seus dados.

```

=====
--          PACKAGE ESPECIFICACION
=====
CREATE OR REPLACE PACKAGE emp_data AS

    TYPE emp_val_cv_type IS REF CURSOR RETURN emp%ROWTYPE;

    PROCEDURE open_emp_cv (emp_cv IN OUT emp_val_cv_type,dept_number IN INTEGER);
    PROCEDURE fetch_emp_data (emp_cv IN emp_val_cv_type, emp_row OUT emp%ROWTYPE);

END emp_data;
/

```

```

=====
--          PACKAGE BODY
=====
CREATE OR REPLACE PACKAGE BODY emp_data AS

=====
PROCEDURE open_emp_cv (emp_cv IN OUT emp_val_cv_type, dept_number IN INTEGER) IS

BEGIN
    OPEN emp_cv FOR SELECT * FROM emp WHERE deptno = dept_number;
END open_emp_cv;

=====
PROCEDURE fetch_emp_data (emp_cv IN emp_val_cv_type, emp_row OUT emp%ROWTYPE) IS
BEGIN
    FETCH emp_cv INTO emp_row;
END fetch_emp_data;

END emp_data;
/

```

Agora um bloco PL/SQL que utiliza as procedures da package para realizar consultas.

```

DECLARE
    -- Declara uma variável cursor do tipo definido na package
    emp_curs emp_data.emp_val_cv_type;

    dept_number dept.deptno%TYPE;
    emp_row emp%ROWTYPE;

```

```

BEGIN
  dept_number := 20;

  -- "Abre" o cursor usando uma variável
  emp_data.open_emp_cv(emp_curs, dept_number);

  -- Extrai os dados e os exibe
  LOOP
    emp_data.fetch_emp_data(emp_curs, emp_row);
    EXIT WHEN emp_curs%NOTFOUND;
    DBMS_OUTPUT.PUT(emp_row.ename || ' ');
    DBMS_OUTPUT.PUT_LINE(emp_row.sal);
  END LOOP;
END;
/

```

Esta forma de implementação permite a reutilização do código da package em vários outros blocos e procedures sem a necessidade de se saber como está definida a consulta. Desta forma, se houver a necessidade de se alterar, por exemplo, a cláusula WHERE da consulta associada ao cursor, apenas o código da package necessita ser alterado, os blocos e procedures que utilizarem esta package não precisam sofrer alteração.

### Exemplo3

O exemplo seguinte mostra a passagem de variáveis do tipo cursor como parâmetro à procedures. É criada uma package, contendo apenas os tipos cursores, e duas procedures. A primeira procedure recebe como parâmetro uma variável do tipo cursor e a quantidade de linhas a serem extraídas e exibe o nome do funcionário e seu salário, a segunda procedure que aceita os mesmos parâmetros da primeira exibe o nome, a função e o salário.

```

CREATE OR REPLACE PACKAGE TIPOS AS

  TYPE emp_tipo_cur IS REF CURSOR RETURN EMP%ROWTYPE;
  TYPE dept_tipo_cur IS REF CURSOR RETURN DEPT%ROWTYPE;

END TIPOS;
/

CREATE OR REPLACE PROCEDURE EXIBE_SALARIO (emp_vcursor_in IN tipos.emp_tipo_cur,
                                           quant_in IN NUMBER) IS

  emp_rec EMP%ROWTYPE;

BEGIN
  FOR indice IN 1 .. quant_in LOOP
    FETCH emp_vcursor_in INTO emp_rec;
    EXIT WHEN emp_vcursor_in%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_rec.ename||' '||emp_rec.sal);
  END LOOP;
END EXIBE_SALARIO;
/

CREATE OR REPLACE PROCEDURE EXIBE_COMPLETO(emp_vcursor_in IN tipos.emp_tipo_cur,
                                           quant_in IN NUMBER) IS

```

```

emp_rec EMP%ROWTYPE;

BEGIN
  FOR indice IN 1 .. quant_in LOOP
    FETCH emp_vcursor_in INTO emp_rec;
    EXIT WHEN emp_vcursor_in%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_rec.ename||' '||emp_rec.job||' '||emp_rec.sal);
  END LOOP;
END EXIBE_COMPLETO;
/

```

Depois de criadas a package e as procedures (usando @nome\_arquivo) o exemplo pode ser testado com o seguinte bloco PL/SQL:

```

DECLARE
  emp_cursor tipos.emp_tipo_cur;
BEGIN
  OPEN emp_cursor FOR SELECT * FROM EMP ORDER BY SAL DESC;
  EXIBE_SALARIO(emp_cursor,4);
  EXIBE_COMPLETO(emp_cursor,5);
  CLOSE emp_cursor;
END;

```

Quando a segunda procedure começa a extrair os dados, ela começa a partir do ponto onde a primeira procedure parou. O mesmo aconteceria com qualquer procedure se o bloco PL/SQL já tivesse extraído algum dado antes de chama-la.

---

### Package UTL\_FILE

---

A package UTL\_FILE permite que programas PL/SQL acessem arquivos do sistema operacional para leitura e escrita. A package pode ser utilizada tanto em programas armazenados no banco de dados quanto em aplicações do lado cliente, tais como as escritas com o Oracle Forms. Desta forma é possível interagir com os arquivos da estação de trabalho e ao mesmo tempo com os arquivos do disco do servidor.

A package UTL\_FILE está disponível com o PL/SQL a partir da versão 2.3

#### O Tipo FILE\_TYPE

Quando você abre um arquivo, o PL/SQL retorna um handle que será utilizado em seu programa. Este handle é do tipo FILE\_TYPE.

FILE\_TYPE é um registro PL/SQL cujos campos possuem todas as informações necessárias ao UTL\_FILE, tais como o nome do arquivo, sua localização e o modo para o qual ele foi aberto.

Uma declaração de um handle é da seguinte forma:

```

DECLARE
  file_handle UTL_FILE.FILE_TYPE;
BEGIN
  ...

```

O acesso as arquivos do sistema é restrito a aqueles situados em diretórios específicos. Uma lista de diretórios acessíveis está armazenada na forma de um parâmetro do arquivo init.ora O acesso não é recursivo aos subdiretórios.

#### A Procedure FCLOSE

FCLOSE é usada para fechar um arquivo aberto, sua especificação é:

```
PROCEDURE FCLOSE(FILE_IN IN UTL_FILE.FILE_TYPE);
```

#### A Procedure FCLOSE\_ALL

Esta procedure fecha todos os arquivos abertos. A especificação para ela é:

```
PROCEDURE FCLOSE_ALL;
```

A procedure FCLOSE\_ALL será útil quando você tiver aberto vários arquivos e desejar ter certeza de que todos foram fechados antes de terminar o programa.

Outra utilização da procedure FCLOSE\_ALL é no tratamento de exceptions, garantindo que os arquivos sejam fechados mesmo que o programa termine de uma forma irregular.

#### A Procedure FFLUSH

A procedure FFLUSH repassa o conteúdo do buffer UTL\_FILE para o arquivo especificado. Este procedimento garante que todas as informações do buffer sejam escritas no arquivo antes que se faça uma leitura. Sua especificação é:

```
PROCEDURE FFLUSH (FILE_IN IN UTL_FILE.FILE_TYPE);
```

#### A Função FOPEN

A função FOPEN abre o arquivo especificado e retorna um handle que deve ser utilizado para manipular o arquivo. A especificação para a função é:

```
FUNCTION FOPEN
  (LOCATION IN VARCHAR2,
   FILE_NAME IN VARCHAR2,
   FILE_MODE IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

O parâmetro "LOCATION" é uma string que especifica o diretório onde se encontra o arquivo.

- Para o DOS / Windows:

```
'C:\COMMON\DEBUG'
```

- Para o Unix:

```
'/usr/admin'
```

O parâmetro "FILE\_NAME" especifica o nome do arquivo a ser aberto

O parâmetro "FILE\_MODE", que especifica o modo de abertura do arquivo, pode ser um dos três seguintes:

R - Abre o arquivo somente para leitura;

W - Abre o arquivo para leitura e sobrescrição, neste modo todas as linhas são removidas;

A - Abre um arquivo para leitura e adição. As novas linhas são adicionadas no final ao arquivo.

#### A Procedure GET\_LINE

Esta procedure lê uma linha de dados do arquivo especificado. Os dados lidos são armazenados na variável fornecida como parâmetro. A especificação para a procedure é:

```
PROCEDURE GET_LINE
  (FILE_IN IN UTL_FILE.FILE_TYPE,
   LINE OUT VARCHAR2);
```

O parâmetro "LINE" deve ser grande o suficiente para receber todos os dados até o primeiro sinal de retorno de carro (carriage return) ou de fim de arquivo, caso contrário será gerada a exception "VALUE\_ERROR".

Se a procedure tentar ler depois do fim do arquivo será gerada a exception "NO\_DATA\_FOUND".

#### A Função IS\_OPEN

A função IS\_OPEN retorna TRUE se o arquivo especificado estiver aberto e FALSE caso contrário. Sua especificação é:

```
FUNCTION IS_OPEN (FILE_IN IN UTL_FILE.FILE_TYPE)
RETURN BOOLEAN;
```

#### Procedure NEW\_LINE

Esta procedure insere um ou mais caracteres de nova linha no arquivo especificado, sua especificação é:

```
PROCEDURE NEW_LINE (FILE_IN IN UTL_FILE.FILE_TYPE,
  NUM_LINES IN PLS.INTEGER := 1);
```

onde "NUM\_LINES" é o número de linhas a serem inseridas no arquivo, seu valor default é um.

#### A Procedure PUT

A procedure PUT coloca dados no arquivo especificado. Sua especificação é:

```
PROCEDURE PUT (FILE_IN IN UTL_FILE.FILE_TYPE,
  ITEM_IN IN VARCHAR2);
PROCEDURE PUT (FILE_IN IN UTL_FILE.FILE_TYPE,
  ITEM_IN IN DATE);
PROCEDURE PUT (FILE_IN IN UTL_FILE.FILE_TYPE,
  ITEM_IN IN NUMBER);
PROCEDURE PUT (FILE_IN IN UTL_FILE.FILE_TYPE,
  ITEM_IN IN PLS_INTEGER);
```

A procedure PUT adiciona os dados à linha corrente do buffer UTL\_FILE. Você deve utilizar a procedure NEW\_LINE para forçar que os dados sejam escritos na próxima linha.

#### A Procedure PUTF

A procedure PUTF coloca dados no arquivo especificado, mas ela usa um formato para os dados para interpretar os diferentes elementos a serem escritos para o arquivo. Você pode passar até cinco elementos diferentes para a procedure PUTF. Sua especificação é:

```
PROCEDURE PUTF (FILE_IN IN UTL_FILE.FILE_TYPE,
               FORMAT_IN IN VARCHAR2,
               ITEM1_IN IN VARCHAR2
               [, ITEM2_IN IN VARCHAR2 ... ITEM5_IN IN VARCHAR2]
               );
```

onde FORMAT\_IN é uma string que especifica o formato dos itens no arquivo. Além de texto, o parâmetro FORMAT\_IN pode possuir os seguintes modelos:

%s - coloca o item correspondente no arquivo. Você pode utilizar até 5 modelos %s na string de formatação.

\n - coloca um caracter de nova linha no arquivo. Não há limites para o uso deste modelo na string de formatação.

A procedure aceita apenas itens do tipo VARCHAR2, se você desejar utilizar tipos diferentes será necessário utilizar a procedure TO\_CHAR para converte-los antes.

#### A Procedure PUT\_LINE

Esta procedure escreve dados para um arquivo e adiciona imediatamente ao final dos dados um caracter de nova linha. Sua especificação é:

```
PROCEDURE PUT_LINE (FILE_IN IN UTL_FILE.FILE_TYPE,
                   ITEM_IN IN VARCHAR2);
```

A procedure PUT\_LINE só aceita dados no formato STRING, caso você deseje escrever outros tipos de dados será necessário utilizar a procedure TO\_CHAR para converter os dados.

#### Tratando exceções de I/O

A package UTL\_FILE oferece um conjunto de exceptions que são específicas para a package. Outras exceptions, como NO\_DATA\_FOUND, também devem ser utilizadas.

Um exemplo de tratamento de erros é:

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Leitura depois do fim do arquivo.');
```

```
  WHEN UTL_FILE.INVALID_OPERATION THEN
    DBMS_OUTPUT.PUT_LINE('Não é possível escrever em arquivo
                          de apenas leitura.');
```

```
END;
```

Uma lista das exceptions que podem ser utilizadas, com suas descrições é dada a seguir:

Consulting Tecnologia & Sistemas Ltda  
www.consulting.com.br

NO\_DATA\_FOUND A procedure GET\_LINE tentou ler além do fim do arquivo.  
 UTL\_FILE.INTERNAL\_ERROR Ocorreu um erro interno. A operação não foi completada.  
 UTL\_FILE.INVALID\_FILEHANDLE O filehandle especificado não é válido ou não foi aberto.  
 UTL\_FILE.INVALID\_MODE O modo fornecido em FOPEN não é válido.  
 UTL\_FILE.INVALID\_OPERATION Operação inválida pois o arquivo não existe ou a operação não é compatível ao modo para o qual o arquivo foi aberto.  
 UTL\_FILE.INVALID\_PATH O caminho fornecido não é acessível.  
 UTL\_FILE.READ\_ERROR Erro específico do sistema operacional na leitura do arquivo.

UTL\_FILE.WRITE\_ERROR Erro específico do sistema operacional na tentativa de se escrever para o arquivo.  
 VALUE\_ERROR O texto lido com GET\_LINE é grande demais para caber no buffer especificado.

#### Exemplo de utilização da package UTL\_FILE

```

DECLARE
  file_handle UTL_FILE.FILE_TYPE;
  nome VARCHAR2(30);
  retrieved_buffer VARCHAR2(100);
BEGIN

  -- abre o arquivo para escrita
  -- o arquivo init.ora deve ter o parametro utl_file_dir = C:\TEMP
  file_handle := UTL_FILE.FOPEN('c:\temp','teste.txt','W');

  -- insere uma linha no arquivo definido por file_handle
  UTL_FILE.PUT_LINE(file_handle, 'Linha 1 (um) ');
  SELECT GLOBAL_NAME INTO nome FROM GLOBAL_NAME;
  -- insere outra linha no arquivo definido por file_handle
  UTL_FILE.PUTF (file_handle, 'Coluna Nome tem o valor %s \n', nome);

  -- fecha o arquivo
  UTL_FILE.FCLOSE(file_handle);

  -- abre o arquivo para leitura
  file_handle := UTL_FILE.FOPEN('c:\temp','teste.txt','R');

  -- inicio de um bloco para leitura
  BEGIN
    LOOP
      -- le uma linha do arquivo definido em file_handle e a exhibe
      UTL_FILE.GET_LINE (file_handle, retrieved_buffer);
      DBMS_OUTPUT.PUT_LINE(retrieved_buffer);
    END LOOP;
  EXCEPTION
    -- quando for fim do arquivo
    WHEN NO_DATA_FOUND THEN
      -- fecha o arquivo
      UTL_FILE.FCLOSE(file_handle);
  
```

```

END;
EXCEPTION
  WHEN UTL_FILE.INVALID_PATH THEN
    DBMS_OUTPUT.PUT_LINE('Caminho inválido.');
```

```

    UTL_FILE.FCLOSE(file_handle);
  WHEN UTL_FILE.READ_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Erro durante a leitura.');
```

```

    UTL_FILE.FCLOSE(file_handle);
  WHEN UTL_FILE.WRITE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Erro durante a escrita.');
```

```

    UTL_FILE.FCLOSE(file_handle);
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Erro!!');
```

```

    UTL_FILE.FCLOSE(file_handle);
END;
```

### Outro Exemplo

```

set serveroutput on;
declare
  vLinha_detalhe  VARCHAR2(32767);
  vLinha_Cabecalho VARCHAR2(600);
  id              UTL_FILE.FILE_TYPE;
  vArquivo       VARCHAR2(20) := 'my_file';
  err            VARCHAR2(100);
  num            NUMBER;
BEGIN

  id := UTL_FILE.FOPEN('/CLDBawb','acumerpawb.txt', 'R', 32767 );
  UTL_FILE.FCLOSE(id);

  vLinha_Cabecalho := 'TESTE DE ARQUIVO';
  --
  vArquivo := 'teste.txt';
  id := UTL_FILE.FOPEN('/CLDBawb',vArquivo, 'W', 32767 );
  UTL_FILE.PUT_LINE(id,vLinha_Cabecalho);

  UTL_FILE.FCLOSE(id);
  --
EXCEPTION
  WHEN OTHERS THEN
    err := SQLERRM;
    num := SQLCODE;
    DBMS_OUTPUT.PUT_LINE(err);
    DBMS_OUTPUT.PUT_LINE(num);
    UTL_FILE.FCLOSE(id);
END;
```

/