PL/SQL

- Extensão ao SQL
- Estruturada em blocos
- Permite controlo do fluxo de execução
- Permite integração entre diferentes ferramentas Oracle
- Não permite comandos DDL

PL/SQL combina:

- poder de manipulação de dados do SQL com
- poder de processamento das lp procedimentais

Principais características:

- Variáveis e constantes
- Tipos de dados escalares e estruturados
- Controlo do fluxo de execução
- Funções integradas
- Gestão de cursores
- Processamento de excepções
- Código armazenado na base de dados

Anonymous Blocks

São blocos anónimos que são declarados numa aplicação no local onde devem ser executados, sendo passados em run-time ao interpretador PL/SQL para execução.

Estruturada em **blocos** (unidade lógica, corresponde a um problema ou sub-problema).

DECLARE
--Definição de objectos PL/SQL a utilizar dentro do bloco.
BEGIN
--Acções executáveis
EXCEPTION
--Processamento de excepções.
END;

Os blocos podem ser encadeados.
Os elementos BEGIN e END são obrigatórios e delimitam o conjunto de acções a efectuar.
A secção DECLARE é opcional e é utilizada para definir objectos de PL/SQL, tais como as variáveis referenciadas no bloco ou num bloco encadeado.
A secção EXCEPTION é opcional e é utilizada para captar excepções, e definir acções a tomar quando estas ocorrem.
Todas as instruções PL/SQL são terminadas com ponto e vírgula.

Subprograms

Blocos anónimos com um nome. Podem ser procedimentos ou funções.

Sintaxe básica do PL/SQL

As instruções podem, se necessário, passar de uma linha para a outra, mas as palavras-chave não podem ser divididas.
As unidades léxicas (identificadores, operadores, etc) podem ser separadas por um ou mais espaços ou por outros limitadores que não se confundam com a unidade léxica.
Não se podem usar palavras reservadas como identificadores, excepto se entre aspas.
Os identificadores têm que começar por uma letra e podem ter até 30 caracteres.
Os valores literais do tipo caracter ou data têm que ser indicados entre plicas.
Os literais numéricos podem ser representados por um só valor ou usando notação científica (2E5=200000).
Os comentário podem ser incluídos entre os símbolos /* e */ quando o comentário engloba várias linhas, ou então após – quando o comentário é apenas uma linha.

Limitadores

Operadores

+	- Adição
-	- Subtracção / negação
*	- Multiplicação
/	- Divisão
IS NULL, LIKE, BETWEEN, IN, =, >, <, <>, !=, ^=, <=, >=	- Comparação
**	- Exponenciação

:=	- Atribuição
=>	- Associação
..	- Intervalo
||	- Concatenação
NOT	- Negação lógica
AND	- Conjunção
OR	- Disjunção

Limitadores

(	Expressão ou lista
)	Expressão ou lista
;	Fim de instrução
‘	Cadeia de caracteres
“	Identificador
<<	Etiqueta

>>	Etiqueta
--	Comentário
/*	Comentário
*/	Comentário

Indicadores

%	Atributo
@	Acesso remoto
:	Variável do host

Separadores

,	Itens

Selectores

.	Componente

Declaração de variáveis e constantes

Variáveis

identificador tipo_de_dados [(precisão, escala)] [NOT NULL] [:= expressão];

Constantes

identificador CONSTANT tipo_de_dados [(precisão, escala)] := expressão;

Atribuições

identificador := expressão;

Domínio dos objectos

```
DECLARE

        X integer;
BEGIN
        ...

        DECLARE
                Y integer;
        BEGIN
                ...
        END;
        ...
END;
```

SUBPROGRAMAS

PROCEDURE	*FUNCTION*
TRIGGER	

ESTRUTURAS DE CONTROLO

IF-THEN-ELSE

```
IF condition1 THEN
     statement1;
 ELSE
     IF condition2 THEN
           statement2;
     ELSE
           IF condition3 THEN
               statement3;
           END IF;
     END IF;
END IF;
```

IF-THEN-ELSIF

```
IF condition1 THEN
   statement1;
ELSIF condition2 THEN
   statement2;
ELSIF condition3 THEN
   statement3;
END IF;
```

LOOP

```
LOOP
   sequence_of_statements;
END LOOP;
```

LOOP LABELS

```
<<outer>>
LOOP
   ...
   LOOP
      ...
      EXIT outer WHEN ...  --
exit both loops
   END LOOP;
   ...
END LOOP outer;
```

EXIT

```
LOOP
   …
   IF count > 100 THEN
      EXIT;
   END IF;
   …
END LOOP;
```

EXIT-WHEN

```
LOOP
   …
   EXIT WHEN count > 100;
   …
END LOOP;
```

WHILE-LOOP

```
WHILE condition LOOP
   sequence_of_statements;
END LOOP;
```

FOR-LOOP

```
FOR counter IN [REVERSE]
lower_bound..higher_bound LOOP
   sequence_of_statements;
END LOOP;
```

CURSORES

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

cursor_parameter_name [IN] datatype [{:= | DEFAULT} expr]
```

```
DECLARE
   CURSOR c1 IS SELECT ename, job
FROM emp WHERE sal < 3000;
   my_record c1%ROWTYPE;
   ...
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO my_record;
      EXIT WHEN c1%NOTFOUND;
      -- process data record
   END LOOP;
   CLOSE c1;
   ...
END;
```

```
DECLARE
   CURSOR c1 (name VARCHAR2,
salary NUMBER) IS SELECT ...
BEGIN
   OPEN c1('ATTLEY', 1500);
   ...
END;
```

EXCEPÇÕES

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
   SELECT price / earnings INTO pe_ratio FROM stocks
      WHERE symbol = 'XYZ';  -- pode causar division-by-zero error
      INSERT INTO stats (symbol, ratio) VALUES ('XYZ', pe_ratio);
EXCEPTION
   WHEN ZERO_DIVIDE THEN  -- trata 'division by zero' error
      INSERT INTO stats (symbol, ratio) VALUES ('XYZ', NULL);
      ...
   WHEN OTHERS THEN  -- handles all other errors
      …;
END;
```

Exemplo de programa:

```
DECLARE
   qty_on_hand  NUMBER(5);
BEGIN
   SELECT quantity INTO qty_on_hand FROM inventory
      WHERE product = 'TENNIS RACKET'
   IF qty_on_hand > 0 THEN  -- check quantity
      UPDATE inventory SET quantity = quantity - 1
         WHERE product = 'TENNIS RACKET';
      INSERT INTO purchase_record
         VALUES ('Tennis racket purchased',
SYSDATE);
   ELSE
      INSERT INTO purchase_record
         VALUES ('Out of tennis rackets',
SYSDATE);
   END IF;
   COMMIT;
END;
```

Comentários:

/* comentário */ ou
-- comentário até ao fim da linha

Tipos de dados:

VARCHAR(n)

Conjunto de caracteres (string) de tamanho variável. *n* varia entre 1 e 2000 caracteres.

VARCHAR2(n)

Conjunto de caracteres (string) de tamanho variável. *n* varia entre 1 e 4000 caracteres.

NUMBER(*p*, *e*)
Representa um número com uma precisão de *p* e uma escala de *e.*

LONG
Conjunto de caracteres de tamanho variável até 2 gigabytes.

BOOLEAN
Valor binário

DATE
Data

CHAR(*n*)
Conjunto de caracteres de tamanho fixo. *n* máximo é de 255 bytes e o comprimento por omissão é de 1 byte

BLOB, CLOB, NCLOB e **BFILE**
tipos de dados para conteúdos binários até 4 Gigabytes internos ou externos (BFILE) à base de dados.

RAW(*n*)
Dados binários em bruto de comprimento variável. *n* máximo é de 255 bytes.

LONG RAW
Dados binários em bruto com um comprimento variável e de tamanho máximo igual a 2 gigabytes.

ROWID
String hexadecimal que representa o endereço único de uma linha numa tabela.

Tipos de dados definidos pelo utilizador:

```
DECLARE
 TYPE TimeRec IS RECORD (minutes SMALLINT, hours
SMALLINT);
 TYPE MeetingTyp IS RECORD (
    day      DATE,
    time     TimeRec,  -- nested record
    place    VARCHAR2(20),
    purpose VARCHAR2(50));
```

Declarar variáveis:

```
part_no  NUMBER(4);
in_stock BOOLEAN;
```

%TYPE e %ROWTYPE:

```
alunos aluno%ROWTYPE;
nomeal4 aluno.nome%TYPE;
```

Declarar constantes:

```
credit_limit CONSTANT REAL := 5000.00;
```

Instrução de atribuição:

1. operador :=

```
bonus := current_salary * 0.10;
```

2. atribuir valor com SELECT ou FETCH:

```
SELECT sal * 0.10
      INTO bonus
      FROM emp
    WHERE empno = emp_id;
```

Declarar cursores:

```
DECLARE
   CURSOR c1 IS
      SELECT empno, ename, job FROM emp WHERE
deptno = 20;
```

Comandos **OPEN, FETCH** e **CLOSE** permitem operar o cursor

```
DECLARE
   CURSOR c1 IS SELECT ename, sal, hiredate,
job FROM emp;
   emp_rec c1%ROWTYPE;

...
OPEN c1
...
FETCH c1 INTO emp_rec;
...
emp_rec.sal := emp_rec.sal * 1.05 - aumento de
5%
```

Estruturas de controlo de fluxo:

1 - Condicional:

```
IF acct_balance >= debit_amt THEN
   UPDATE accounts SET bal = bal - debit_amt
      WHERE account_id = acct;
ELSE
   INSERT INTO temp VALUES
      (acct, acct_balance, 'Insufficient funds');
END IF;
```

2 - Iterativo:

```
LOOP
    -- sequence of statements
END LOOP;
```

FOR-LOOP

```
FOR i IN 1..order_qty LOOP
  UPDATE sales SET custno = customer_id
     WHERE serial_num =
serial_num_seq.NEXTVAL;
END LOOP;
```

WHILE-LOOP

```
WHILE salary < 4000 LOOP
     SELECT sal, mgr, ename INTO salary,
mgr_num, last_name
      FROM emp WHERE empno = mgr_num;
END LOOP;
```

EXIT WHEN

```
LOOP
   ...
   total := total + salary;
    EXIT WHEN total > 25000;  -- exit
loop if true
END LOOP;
```

3 – Sequencial:

```
IF rating > 90 THEN
    GOTO calc_raise;  -- branch to label
END IF;
...
<<calc_raise>>
IF job_title = 'SALESMAN' THEN
```

```
      amount := commission * 0.25;
   ELSE
      amount := salary * 0.10;
   END IF;
```

Modularidade:

1. Subprograms:

Procedures (para realizar acções)

```
PROCEDURE award_bonus (emp_id NUMBER)
IS
   bonus          REAL;
   comm_missing EXCEPTION;
BEGIN
   SELECT comm*0.15 INTO bonus
      FROM emp WHERE empno = emp_id;
   IF bonus IS NULL THEN
      RAISE comm_missing;
   ELSE
      UPDATE payroll SET pay = pay +
bonus
            WHERE empno = emp_id;
   END IF;
EXCEPTION
   WHEN comm_missing THEN
      ...
END award_bonus;
```

Functions (para calcular e retornar um valor)

```
FUNCTION sal_ok (salary REAL, title
```

```
REAL) RETURN BOOLEAN IS
   min_sal REAL;
   max_sal REAL;
BEGIN
   SELECT losal, hisal INTO min_sal,
max_sal
      FROM sals
      WHERE job = title;
   RETURN (salary >= min_sal) AND
(salary <= max_sal);
END sal_ok;
```

2. **External Procedures**
3. **Packages**

Tratamento de excepções:

Excepção: condição de erro; quando ocorre o erro é levantada uma excepção que interrompe o fluxo normal de execução do programa e o direcciona para uma rotina de tratamento de excepções (***exception handler***)

Excepções **pré-definidas** são levantadas implicitamente pelo sgbd:

- **CURSOR_ALREADY_OPEN** -> tentativa de abrir um cursor já aberto
- **INVALID_CURSOR** -> aceder a um cursor que não está aberto
- **INVALID_NUMBER** -> conversão inválida de uma string num numero

- **NO_DATA-FOUND** -> o comando select não retornou nenhuma linha
- **VALUE_ERRORS** -> conversão de tipos sem sucesso ou atribuição de valores superiores à suportada pela variável
- **TOO_MANY_ROWS** -> comando select retornou mais do que uma linha
- **ZERO_DIVIDE** -> divisão por zero

Excepções **definidas pelo utilizador** têm que ser declaradas e são levantadas com o comando **RAISE**

```
DECLARE
   ...
   comm_missing EXCEPTION;  -- declare
exception
BEGIN
   ...
   IF commission IS NULL THEN
      RAISE comm_missing;  -- raise exception
   ELSE
      bonus := (salary * 0.10) + (commission * 0.15);
   END IF;
EXCEPTION
   WHEN comm_missing THEN
        -- process error    -- exception
handler
```

PROCEDURE

Para realizar uma determinada acção

Síntaxe:

```
PROCEDURE name [(parameter[,
parameter, ...])] IS
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception handlers]
END [name];
```

Especificação de parâmetros:

```
parameter_name [IN | OUT | IN OUT]
datatype [{:= | DEFAULT} expression]
```

Exemplo:

```
PROCEDURE raise_salary (emp_id INTEGER,
increase REAL) IS
   current_salary REAL;
   salary_missing EXCEPTION;
BEGIN
   SELECT sal INTO current_salary FROM emp
      WHERE empno = emp_id;
   IF current_salary IS NULL THEN
      RAISE salary_missing;
   ELSE
      UPDATE emp SET sal = sal + increase
         WHERE empno = emp_id;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO emp_audit VALUES (emp_id,
'No such number');
```

```
   WHEN salary_missing THEN
      INSERT INTO emp_audit VALUES (emp_id,
'Salary is null');
END raise_salary;
```

Um procedure é invocado como um comando PL/SQL:

```
raise_salary(emp_num, amount);
```

FUNCTION

Para calcular e retornar um valor

Síntaxe:

```
FUNCTION name [(parameter[,
parameter, ...])] RETURN datatype IS
   [local declarations]
BEGIN
   executable statements
[EXCEPTION
   exception handlers]
END [name];
```

Especificação de parâmetros:

```
parameter_name [IN | OUT | IN OUT]
datatype [{:= | DEFAULT} expression]
```

A síntaxe de uma *function* é idêntica à de um *procedure* mas a *function* contém uma cláusula RETURN que especifica o tipo de dados de retorno

Exemplo:

```
FUNCTION sal_ok (salary REAL, title REAL)
RETURN BOOLEAN IS
   min_sal REAL;
   max_sal REAL;
BEGIN
   SELECT losal, hisal INTO min_sal,
max_sal
      FROM sals
      WHERE job = title;
```

```
   RETURN (salary >= min_sal) AND (salary
<= max_sal);
END sal_ok;
```

Uma *function* é invocada como parte de uma expressão; o identificador de função actua como uma variável cujo valor depende dos parâmetros:

```
IF sal_ok(new_sal, new_title) THEN ...
```

Instrução **RETURN**: termina a execução de um sub-programa e retorna o controlo para o ponto de chamada

- Cada sub-programa pode ter várias instruções RETURN (embora seja má prática de programação)
- Num PROCEDURE não podem ter uma expressão associada
- Numa FUNCTION têm que ter uma expressão associada
- Uma FUNCTION tem que ter obrigatoriamente uma instrução RETURN, caso contrário o PL/SQL levanta a excepção PROGRAM_ERROR

Stored subprograms

Para criar sub-programas e armazená-los permanentemente numa bd:

CREATE PROCEDURE ... e **CREATE FUNCTION ...**

Parâmetros

Formais (declarados na definição do sub-programa) e **reais** (variáveis passadas na invocação)

Chamada por posição e nome

```
PROCEDURE credit_acct (acct_no INTEGER, amount
REAL) IS ...

credit_acct(acct, amt);                    --
positional notation
credit_acct(amount => amt, acct_no => acct);
-- named notation
credit_acct(acct_no => acct, amount => amt);
-- named notation
credit_acct(acct, amount => amt);
-- mixed notation
```

MODOS/COMPORTAMENTO

IN – internamente são como constantes, não podem ser alterados no sub-programa (modo por defeito), passagem por referência

OUT – para retornar valores

IN OUT – permite a passagem de valores para o sub-programa e o retorno, passagem por valor

IN	OUT	IN OUT
the default	must be specified	must be specified
passes values to a subprogram	returns values to the caller	passes initial values to a subprogram and returns updated values to the caller
formal parameter acts like a constant	formal parameter acts like an uninitialized variable	formal parameter acts like an initialized variable
formal parameter cannot be assigned a value	formal parameter cannot be used in an expression and must be assigned a value	formal parameter should be assigned a value
actual parameter can be a constant, initialized variable, literal, or expression	actual parameter must be a variable	actual parameter must be a variable
actual parameter is passed by reference (a pointer to the value is passed in)	actual parameter is passed by value (a copy of the value is passed out)	actual parameter is passed by value (a copy of the value is passed in and out)

Overloading:

Os identificadores/nomes de sub-programas podem ser usados para definir sub-programas diferentes desde que tenham parâmetros formais diferentes (em número, ordem e/ou família do tipo de dados)

TRANSACÇÕES

Controlo de concorrência: acessos simultâneos ao mesmo objecto

O acesso simultâneo a dados é controlado com mecanismos de **lock**:

tabelas

```
LOCK TABLE emp IN ROW SHARE MODE NOWAIT;
```

ou linhas/registos

```
DECLARE
CURSOR c1 IS SELECT empno, sal FROM emp
WHERE job = 'SALESMAN' AND comm > sal FOR
UPDATE;
```

A primeira instrução de um programa PL/SQL **inicia uma transacção** que decorre até que ocorra um COMMIT ou ROLLBACK; a primeira instrução após COMMIT ou ROLLBACK inicia nova transacção

COMMIT: fecha a transacção em curso e torna definitivas as alterações efectuadas sobre a bd durante essa transacção; liberta todos os locks a tabelas e linhas

ROLLBACK: fecha a transacção em curso e desfaz as alterações efectuadas sobre a bd durante essa transacção; coloca a bd no estado em que estava antes do início da transacção; liberta todos os locks a tabelas e linhas

SAVEPOINT: marca um ponto no código como referência para realizar ROLLBACKs parciais

```
BEGIN
...
SAVEPOINT my_point;
UPDATE emp SET ... WHERE empno = emp_id;
...
SAVEPOINT my_point; -- move my_point to
current point
INSERT INTO emp VALUES (emp_id, ...);
...
```

```
EXCEPTION
WHEN OTHERS THEN
ROLLBACK TO my_point;
END;
```

CURSORES

***1ª Fase** – Declaração do cursor explícito – síntaxe:*

CURSOR cursor_name [(parameter[, parameter]...)]
[RETURN return_type] IS
select_statement;

```
DECLARE
CURSOR c1 IS SELECT empno, ename,
job, sal FROM emp
WHERE sal > 2000;
CURSOR c2 RETURN dept%ROWTYPE IS
SELECT * FROM dept WHERE deptno = 10;
```

***2ª Fase** – Abertura de cursor explícito – síntaxe:*

OPEN cursor_name;

***3ª Fase** – Processamento do result set (um registo de cada vez) – síntaxe:*

FETCH cursor_name
INTO {variable1[, variable2 ...]}
record_name;

```
LOOP
FETCH c1 INTO my_record;
EXIT WHEN c1%NOTFOUND;
-- process data record
```

```
END LOOP;
```

Deve haver correspondência entre número e tipo de dados da declaração do cursor, cláusula SELECT, e da cláusula INTO do FETCH

O número de cursores simultaneamente abertos (OPEN_CURSORS) constitui uma variável de ambiente e encontra-se limitado.

A tentativa de efectuar o fetch de dados de um cursor fechado desencadeia a excepção INVALID_CURSOR

4ª Fase *– Fecho de cursor explícito – síntaxe:*

CLOSE cursor_name;

Cursor FOR LOOP:

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

record_name é um identificador declarado implicitamente como `record_name cursor_name%ROWTYPE`

cursor_name identifica um cursor que **não pode** ter sido aberto

```
DECLARE
result temp.col1%TYPE;
CURSOR c1 IS
SELECT n1, n2, n3 FROM data_table WHERE
expr_num = 1;
BEGIN
```

```
FOR c1_rec IN c1 LOOP
result := c1_rec.n2 / (c1_rec.n1 +
c1_rec.n3);
INSERT INTO temp VALUES (result, NULL,
NULL);
END LOOP;
COMMIT;
END;
```

Passagem de parâmetros:

```
DECLARE
CURSOR emp_cursor(dnum NUMBER) IS
SELECT sal, comm FROM emp WHERE
deptno = dnum;
...
BEGIN
FOR emp_record IN emp_cursor(20) LOOP
...
END LOOP;
...
END;
```

Atributos de cursores explícitos: retornam informação sobre o processamento do cursor.

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
OPEN	**before**	exception	FALSE	exception	exception
	after	NULL	TRUE	NULL	0
First FETCH	**before**	NULL	TRUE	NULL	0
	after	TRUE	TRUE	FALSE	1
Next FETCH(es)	**before**	TRUE	TRUE	FASE	1
	after	TRUE	TRUE	FASE	data dependent

Last FETCH	**before**	TRUE	TRUE	FASE	data dependent
	after	FALSE	TRUE	TRUE	data dependent
CLOSE	**before**	FALSE	TRUE	TRUE	data dependent
	after	exception	FALSE	exception	exception

Notes:

Referencing %FOUND, %NOTFOUND, or %ROWCOUNT before a cursor is opened or after it is closed raises INVALID_CURSOR.

After the first FETCH, if the result set was empty, %FOUND yields FALSE, %NOTFOUND yields TRUE, and %ROWCOUNT yields 0.

Atributos de cursores implícitos: retornam informação sobre o resultado da última instrução INSERT, DELETE, UPDATE ou SELECT INTO executada.
São acedidos através de **SQL%attribute_name**

SQL%FOUND é TRUE se uma instrução INSERT, DELETE ou UPDATE afectou, ou se um SELECT INTO retornou, um ou mais registos (%NOTFOUND é a negação de %FOUND)

SQL%ISOPEN é FALSE uma vez que o Oracle fecha o cursor implícito imediatamente após a execução da instrução

SQL%ROWCOUNT retorna o número de registos afectados por uma instrução INSERT, DELETE ou UPDATE, ou retornados por um SELECT INTO

Ciclos FOR com subqueries

Não é necessário declarar o cursor (o cursor é o próprio select).

Não é possível invocar os atributos de um cursor explícito definido como subquery de um ciclo FOR de

cursor (porque não tem nome).

```
BEGIN
FOR emp_record IN (SELECT ename,
deptno
FROM emp) LOOP
     -- implicit open and implicit fetch
occur
     IF emp_record.deptno = 30 THEN
     ...
END LOOP; -- implicit close occurs
END;
```

TRIGGER

Triggers são procedimentos de PL/SQL que são executados (disparados) quando ocorre um dos seguintes tipos de operações:

- **instruções de DML num objecto schema especifico**
- **instruções de DDL feitos num schema ou numa bd**
- **eventos de Login/Logoff do utilizador**
- **erros de servidor**
- **Startup/Shutdown da bd**

Principal diferença PROCEDURE/TRIGGER

1. **PROCEDURE é executado quando invocado explicitamente pelo utilizador**

2. **TRIGGER é executado quando invocado implicitamente pelo Oracle sempre que um evento de triggering ocorra, independentemente do utilizador ou aplicação que o use**

3. **A utilização de triggers deve ser muito cuidadosa (apenas quando necessário) o uso excessivo de triggers pode resultar em interdependências complexas (*Cascadind Triggers*) que dificultam a manutenção de grandes aplicações.**

Utilização de triggers na restrição de dados de input:

1 - quando uma regra de integridade não pode ser assegurada através de:

- **NOT NULL, UNIQUE**
- **PRIMARY KEY**
- **FOREIGN KEY**
- **CHECK**
- **DELETE CASCADE**
- **DELETE SET NULL**

2 - para assegurar regras de negócio complexas que não é possível impôr através de CONSTRAINTS

3 - para assegurar a integridade referencial quando tabelas "filho" e tabelas"pai" estão em diferentes nós de uma bd distribuida

Um Trigger é composto por 3 partes:

O evento ou instrução de Triggering;
A restrição;
A acção ou corpo;

Quando se define um Trigger é possível especificar se este deve ser executado:

para cada linha afectada pela instrução de triggering, tal como um Update statement que actualiza 'n' linhas. (triggers de linha)

para cada instrução de triggering, independentemente do numero de linhas que afecte (triggers de instrução)

antes da instrução de triggering

depois da instrução de triggering

Exemplo de criação de um trigger:

```
CREATE OR REPLACE TRIGGER Trg_Mostra_Encomenda -- nome do trigger
BEFORE INSERT OR UPDATE ON VendasDetalhes -- instrução de Triggering
FOR EACH ROW
WHEN (new.qtd_encomenda > 0) -- restrição
DECLARE – inicio da acção ou corpo do trigger
v_dif number;
BEGIN
v_dif := :new.qtd_encomenda - :new.qtd_enviada;
dbms_output.put_line ('Trigger TRG_MOSTRA_ENCOMENDA disparou!');
dbms_output.put_line('Quantidade encomendada: ' || :new.qtd_encomenda)
dbms_output.put_line('Quantidade Enviada: ' || :new.qtd_enviada)
dbms_output.put_line(' Quantidade por enviar: ' || v_dif);
END;
```

Controlando o Timming dum Trigger

- A opção BEFORE ou AFTER no CREATE TRIGGER especifica se a acção do trigger deve ser executada ANTES ou DEPOIS da instrução de triggering a ser executada.

Executando o trigger uma ou mais vezes (FOR EACH ROW)

- Esta opção, quando especificada,"dispara" o trigger em cada registo afectado pela instrução de triggering.
- A ausência desta opção indica que o trigger só é executado uma única vez para cada instrução e não separadamente para cada registo afectado

Executando um trigger com base numa condição (cláusula WHEN)

- **Opcionalmente, pode ser incluida uma restrição na definição dum TRIGGER DE LINHA (não num trigger de instrução) especificando uma expressão de SQL booleana na cláusula WHEN (ver exemplo anterior).**
- **Se esta opção for incluida, a expressão será avaliada para todos os registos afectados pelo trigger. Esta expressão quer seja avaliada em TRUE ou FALSE, não implica no rollback da instrução de triggering, ou seja, este último é sempre executado independentemente do resultado da avaliação da expressão.**
- **Se a avaliação resultar em TRUE para o registo, então a acção do trigger é executada em relação a esse registo, caso contrário não será executada.**
- **A expressão na cláusula WHEN deve ser uma expressão SQL e não pode incluir subqueries;**

Tipos de Triggers

	Linha	**Instrução**	**Before**	**After**
Execução	Executado sempre que uma tabela é afectada pela instrução de triggering	Executado tendo em consideração a instrução de triggering, independentemente do número de registos afectados	Executa a acção do trigger antes da instrução de triggering;	Executa a acção do trigger depois de executada a instrução de triggering
Utilidade	Se o código contido na acção do trigger depender dos dados resultantes da instrução de triggering ou dos registos afectados	Se o código na acção do trigger não depender dos dados resultantes da instrução de triggering ou dos registos afectados	Permite eliminar processamento desnecessário da instrução de triggering e o seu eventual rollback (casos em que se geram excepções na acção do trigger)	Controlar o timming dum trigger;
Aplicação		Questões de segurança relacionadas com o utilizador; Registos de Auditoria;	Cálculos de valores de colunas específicas antes da instrução de triggering (INSERT ou DELETE) estar completa	

Tipo Trigger	Caracteristicas
BEFORE instrução	A acção do trigger é executada antes da instrução de triggering;
AFTER instrução	A acção do trigger é executada depois de executada a instrução de triggering
BEFORE linha	A acção do trigger é executada: • de acordo com a restrição do trigger; • antes de cada linha ser afectada pela instrução de triggering; • antes da verificação das restrições de integridade.
AFTER linha	A acção do trigger é executada para cada registo de acordo com a restrição do trigger e depois de modificados os registos pela instrução de triggering.É feito o lock dos registos afectados.

É possivel ter vários triggers do mesmo tipo para a mesma tabela.

Aceder aos valores dos campos nos TRIGGER DE LINHAS

Nomes de correlação

No corpo dum trigger é possível aceder aos valores antigos e novos dos campos do registo afectado pela instrução de triggering.

Existem dois nomes de correlação para cada coluna da tabela a ser modificada:

- um para o valor antigo (**:OLD**) e
- outro para o valor novo (**:NEW**):

Operação	:OLD (utilidade)	:NEW (utilidade)
INSERT	X	√
UPDATE	√	√
DELETE	√	X

Detectar a operação DML que "disparou" o trigger (INSERTING, UPDATING, e DELETING)

Mais do que um tipo de operação DML pode disparar um trigger (por exemplo: ON INSERT OR DELETE OR UPDATE of Editoras).

Na acção do trigger utilizam-se predicados condicionais (INSERTING, DELETING e UPDATING) para verificar qual dos tipos de operação "disparou" o trigger.

```
IF INSERTING THEN ... END IF; -- TRUE se
foi um INSERT que "disparou" o trigger
```

```
IF UPDATING THEN ... END IF; -- TRUE se
foi um UPDATE que "disparou" o trigger
```

Num trigger de UPDATE pode-se especificar o nome do campo a ser actualizado.

```
CREATE OR REPLACE TRIGGER ...
```

```
... UPDATE OF qtd_encomenda, qtd_enviada ON
VendasDetalhes ...
```

```
BEGIN
```

```
... IF UPDATING ('qtd_encomenda') THEN
... END IF;
```

```
END;
```

O código da estrutura de controlo IF só será executado se a instrução de triggering actualizar a coluna 'qtd_encomenda'.

Manipulação de Triggers

Apagar um trigger:

2. **DROP TRIGGER** <nome_trigger>;

2.

- Alterar o estado (ACTIVO/DESACTIVO) dum trigger:

1.

ALTER TRIGGER <nome_trigger> ENABLE | DISABLE [ALL TRIGGERS];

Condições de Erro e Excepções num Trigger

- Qualquer condição de erro ou excepção ocorrida durante a execução dum trigger provoca o roollback da instrução de triggering e do corpo do trigger excepto quando são criados exception handlers.
- As excepções definidas pelo utilizador são as mais usadas nos triggers que pretendem assegurar a integridade das restrições.

Utilizando Excepções em Triggers

- Tal como nos sub-programas, também pode ser feito o tratamento de excepções em triggers.

```
CREATE OR REPLACE
TRIGGER trg_salarios2
BEFORE UPDATE OF salary ON employee
FOR EACH ROW
     DECLARE
          too_much EXCEPTION;
     BEGIN
          IF :NEW.salary>99000 THEN
               RAISE too_much;
          END IF;
          EXCEPTION
               WHEN too_much THEN
               RAISE_APPLICATION_
               ERROR (-
               20001,'Cannot pay
               that much');
               END;
```