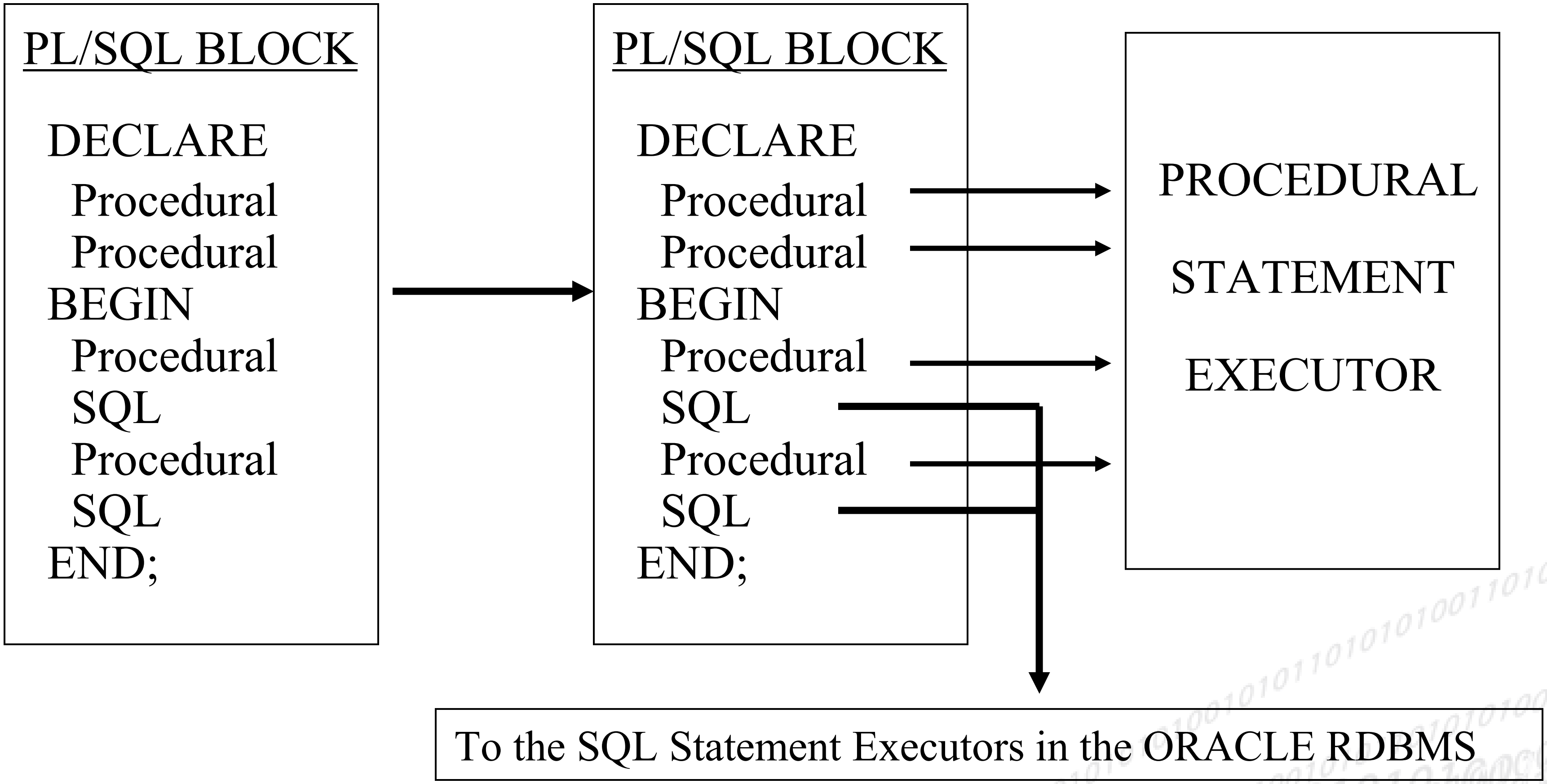
# Oracle 9i
# PL / SQL

**Draft**

# PL/SQL

## Introduction

# PL/SQL Execution  Environments - The PL/SQL Engine



PL/SQL BLOCK

```
DECLARE
  Procedural
  Procedural
BEGIN
  Procedural
  SQL
  Procedural
  SQL
END;
```

PL/SQL BLOCK

```
DECLARE
  Procedural
  Procedural
BEGIN
  Procedural
  SQL
  Procedural
  SQL
END;
```

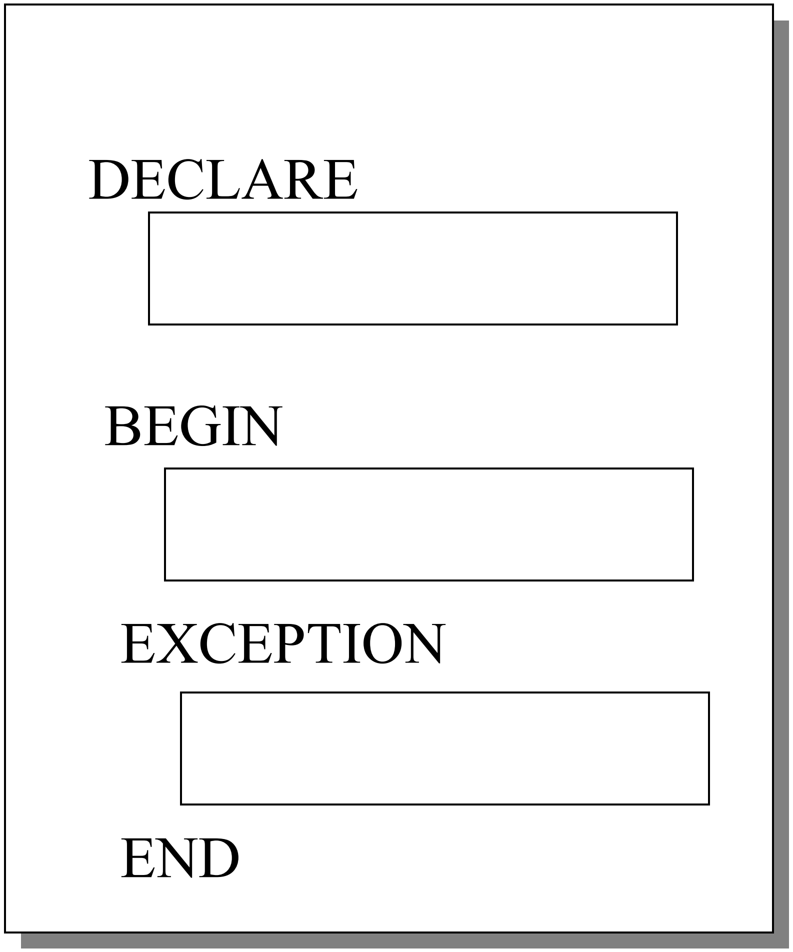
PROCEDURAL

STATEMENT

EXECUTOR

To the SQL Statement Executors in the ORACLE RDBMS

# PL/SQL BLOCK STRUCTURE



DECLARE

BEGIN

EXCEPTION

END

# PL/SQL BLOCK STRUCTURE

**Declaration Section**

**Executable Section**

**Exception Handler Section**

Quick Note - **Block structuring**

1. Any block may contain sub-block. Sub-blocks may appear anywhere an executable statement may legally appear.

2. Statements end with a semi-colon (;)

3. Comments are preceded by -- or surrounded by /* */

4. Declared objects exist within a certain scope

# PL/SQL

## Variable Declaration

# Variable Declarations Overview

Syntax of Declaration

identifier [constant ] datatype [not null ] [:= plsql_expression ] ;

Quick Notes - Variable Declaration

1. The rules for identifiers are same as for SQL objects.
2. NOT NULL/CONSTANT may be optionally used
3. Only one identifier per line is allowed .

```
DECLARE
        firstname        lastname  CHAR(20)  ; - illegal
DECLARE
        firstname        CHAR(20)  ; -legal
        lastname         CHAR(20)  ; - legal
```

# Variable Declarations Overview

**NUMBER**

```
Count                  NUMBER;
revenue                NUMBER (9,2);
second_per_day         CONSTANT NUMBER := 60 * 60* 24 ;
running _total         NUMBER (10,0) := 0;
```

**VARCHAR2**

```
mid_initial            VARCHAR2 := ‘K’;
last_name              VARCHAR2(10) NOT NULL;
company_name           CONSTANT VARCHAR2(12);
```

**DATE**

```
anniversary            DATE := ‘05-NOV-78’;
project_complexion     DATE;
next_checkup           DATE NOT NULL := ‘28-JUN-90’;
```

**BOOLEAN**

```
over_budget            BOOLEAN   NOT NULL   := FALSE;
available              BOOLEAN   := NULL ;
```

# Attribute Declaration

PL/SQL objects (such as variables and constants) and database objects (such as columns and tables ) are associated with certain attributes.

**%TYPE attribute**

```
DECLARE
  books_printed          NUMBER (6);
  books_sold             books.sold%TYPE ;
  maiden_name            emp.ename%TYPE ;
```

**%ROWTYPE attribute**

```
DECLARE
  dept_row               dept%ROWTYPE ;
```

# Variable Assignment

**PL/SQL Expressions** consist of Variables, Constants, Literals, and Function Calls.

**ASSIGNMENT** **Syntax**

plsql_variable := plsql_expression;

**Quick notes -Assignment**

**1**. := (ASSIGNMENT ) whereas = (VALUE EQUALITY)

**2**. The datatype of the left and right hand side of an assignment must be the same or implicitly convertible to each other.

***For ex***. *, N:=‘7’ is legal because number may be implicitly converted to char.*

**3**. Column or table reference are not allowed on either side of an assignment operator( : = ).

```
SCOTT.EMP.EMPNO := 1234;
location := dept.loc.;
```

} **These are illegal**

# Scoping

**SCOPE** refers to the visibility of identifiers at different points in the PL/SQL block.

## SCOPING RULES:

1. An identifier is visible in the block in which it is declared and all its sub-blocks unless rule #2 applies.

2. If an identifier in an enclosing block is redeclared in a sub-block, the original identifier declared in the enclosing block is no longer visible in the sub-block .However, the newly declared identifier has the rules of scope defined in rule #1.

# Scoping Variables and Constants

```
DECLARE
credit_limit    CONSTANT NUMBER (6,2) : =2000;
account          NUMBER := 100;
BEGIN
     DECLARE
        account          CHAR(10) := 'ABC';
        new_balance    NUMBER (9,2);
     BEGIN
           new_balance    account    credit_limit
     END;
     DECLARE
        account          NUMBER := 200;
        old_balance    NUMBER (9,2);
     BEGIN
           old_balance    account    credit_limit
     END;
END;        account        credit_limit
```

# PL/SQL

## SQL in PL/SQL

# SQL & PL/SQL Overview

**SQL Data Manipulation Language** (DML) statement support

1. INSERT
2. UPDATE
3. DELETE
4. SELECT

**QuickNotes - SQL DML Support**

1. The full ORACLE syntax is supported for these statements

2. A PL/SQL variable may be placed anywhere a constant may be legally placed.

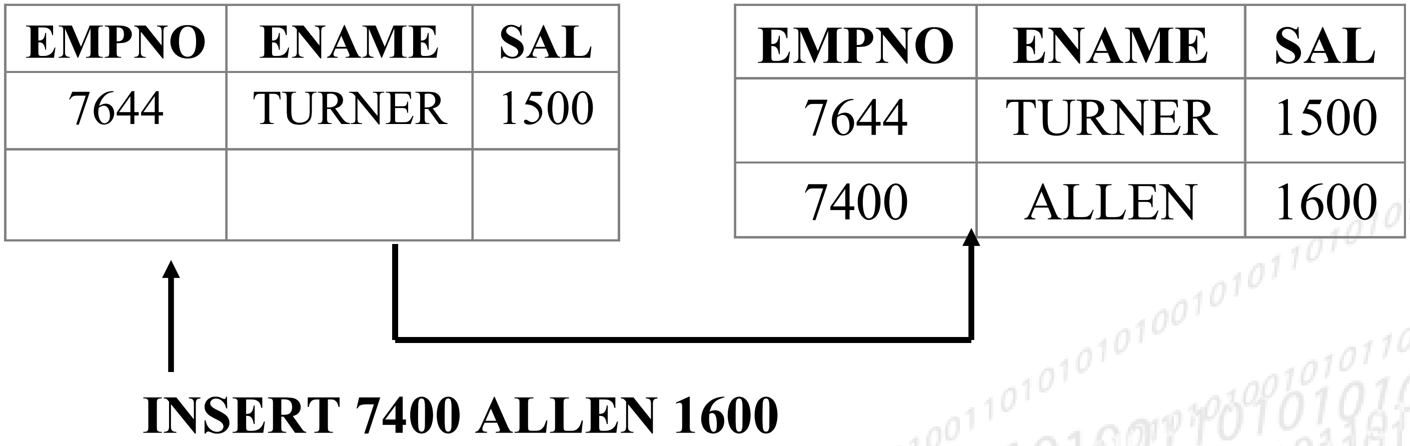
3. An identifier is first checked to see if it is a column in the database. If not, it is assumed to be a PL/SQL identifier.

4. These statements may not appear as part of an expression

# SQL & PL/SQL Overview

## INSERT

```
DECLARE
   my_sal    NUMBER(7,2)  := 3040.22;
   my_ename   CHAR(25)  := ‘WANDA’;
   my_hiredate  DATE := ‘08-SEP-01’;
BEGIN
   INSERT INTO  EMP (empno, ename, job, hiredate, sal , deptno)
   VALUES (2345, my_ename, ’cab Driver’, my_hiredate, my_sal, 20);

END;
```



| EMPNO | ENAME | SAL |
|---|---|---|
| 7644 | TURNER | 1500 |
| | | |

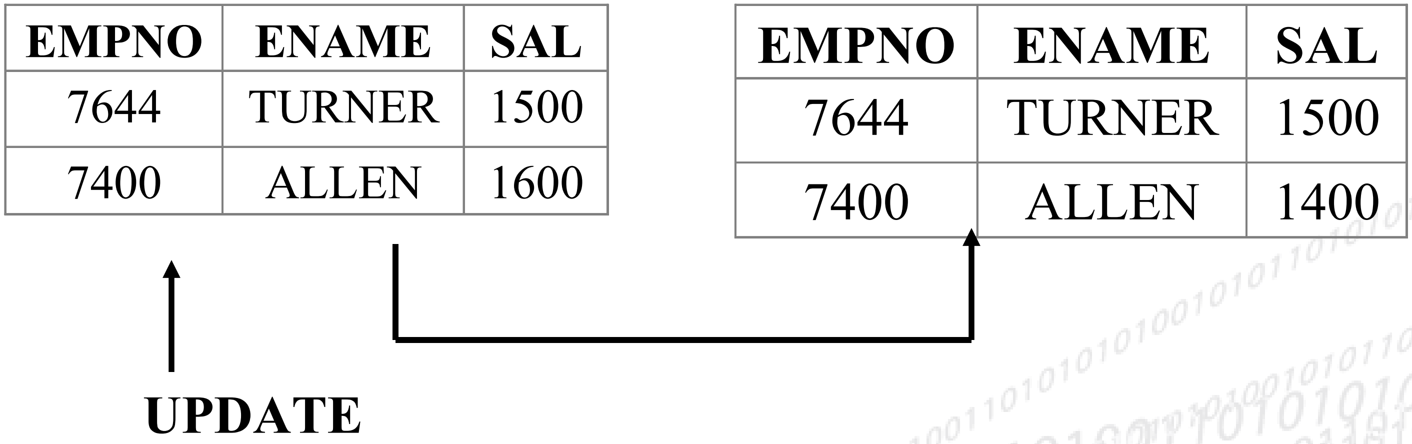
| EMPNO | ENAME | SAL |
|---|---|---|
| 7644 | TURNER | 1500 |
| 7400 | ALLEN | 1600 |

**INSERT 7400 ALLEN 1600**

# SQL & PL/SQL Overview

## UPDATE

```
DECLARE
   max_allowed    CONSTANT N UMBER  := 5000;
   good_cust        CHAR(8)  := ‘VIP’;
BEGIN
   UPDATE ACCOUNT SET CREDIT_LIMIT = MAX_ALLOWED
   WHERE   TYPE = ‘EMPLOYEE ‘ OR TYPE = good_cust;
END;
```



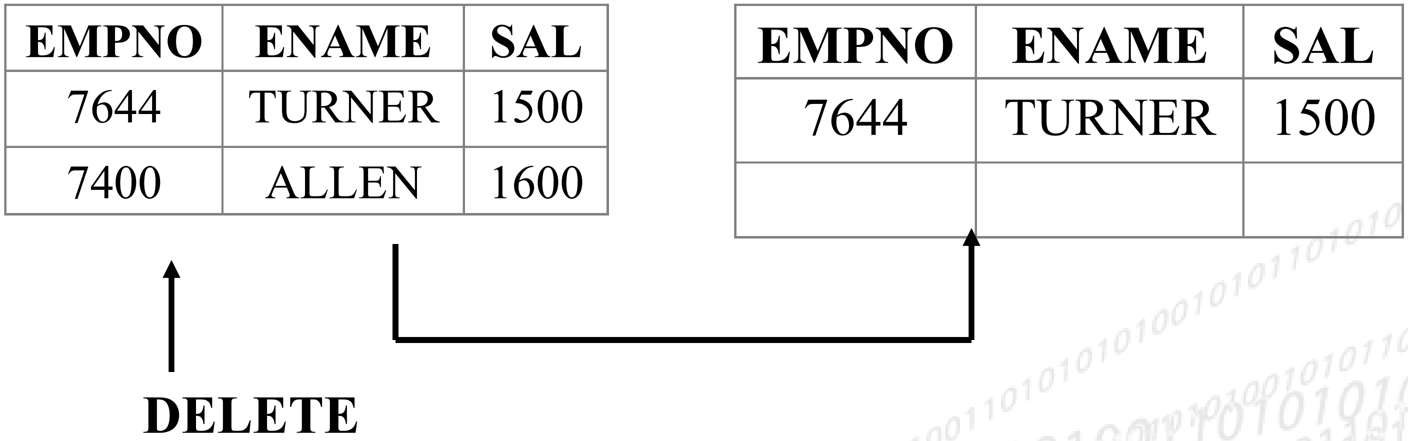
| EMPNO | ENAME | SAL |
|---|---|---|
| 7644 | TURNER | 1500 |
| 7400 | ALLEN | 1600 |

| EMPNO | ENAME | SAL |
|---|---|---|
| 7644 | TURNER | 1500 |
| 7400 | ALLEN | 1400 |

# SQL & PL/SQL Overview

## DELETE

```
DECLARE
  bad_child_type    CHAR(8)  := ‘NAUGHTY’;
BEGIN
  DELETE  FROM  santas_gift_list  WHERE   kid_rating = bad_child_type ;
END;
```

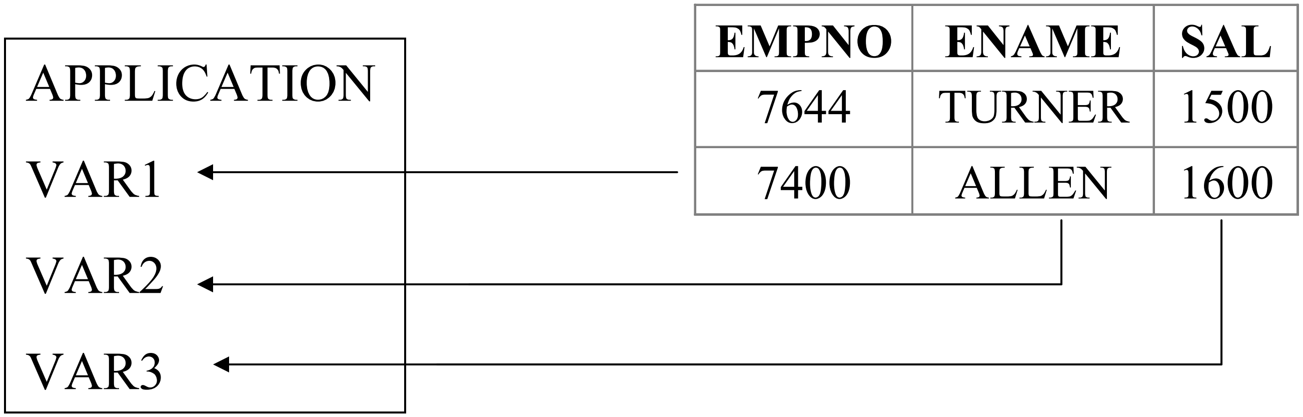


| EMPNO | ENAME | SAL |
|---|---|---|
| 7644 | TURNER | 1500 |
| 7400 | ALLEN | 1600 |

| EMPNO | ENAME | SAL |
|---|---|---|
| 7644 | TURNER | 1500 |
| | | |

DELETE

# SQL & PL/SQL Overview



APPLICATION

VAR1

VAR2

VAR3

| EMPNO | ENAME | SAL |
|---|---|---|
| 7644 | TURNER | 1500 |
| 7400 | ALLEN | 1600 |

## QuickNotes - SELECT INTO

1. A SELECT statement is the only DML that returns data. You must provide location for this data to be stored via the INTO clause.
2. A SELECT..INTO statement must return exactly one row. Zero or multiple returned rows result in an error.
3. For multi-row SELECTs use cursors.

# SQL & PL/SQL Overview

## SELECT Syntax

**SELECT** col1,col2……INTO var1,var2.. FROM   table_name  WHERE ...

**E.g.**

```
DECLARE
   part_name            parts.name%TYPE;
   num_in_stock        parts.num%TYPE;
BEGIN
 SELECT name, num
 INTO part_name, num_in_stock
 FROM PARTS
 WHERE part_id  =  234;
 -- manipulate the retrieved data here
```

# Transaction processing

## SAVEPOINT Syntax

**SAVEPOINT** <marker_name>**;**

## ROLLBACK TO Syntax

ROLLBACK [WORK] TO SAVEPOINT <marker_name>**;**

## SAVEPOINT and ROLLBACK TO Ex.

```
BEGIN
    INSERT INTO temp VALUES (1,1 ‘ROW 1’);   SAVEPOINT A;
    INSERT INTO temp VALUES (2,2 ‘ROW 2’);   SAVEPOINT B ;
    ….
    ROLLBACK TO SAVEPOINT B;
    COMMIT ;
END;
```

# SQL Functions

**SQL Functional Support** (within a SQL Statement):

1. Numeric (e.g. SQRT, ROUND, POWER)
2. Character (e.g. LENGTH, UPPER)
3. Date (e.g. ADD_MONTHS, MONTH_BETWEEN);
4. Group(e.g. AVG, MAX, COUNT)

INSERT INTO phonebook (lastname) VALUES (UPPER(my_lastname));

**Other SQL Functional Support** (outside of a SQL Statement):

MOST ORACLE SQL functional are available (except for group functions).

X := SQRT(y);

lastname := UPPER (lastname);

age_diff := MONTHS_BETWEEN(birthday1,birthday2)/12;

# PL/SQL

# Conditional & Iterative Control

# Logical Comparisons

- Logical Comparisons form the basis of conditional control in PL/SQL
- The result of these comparisons are always either TRUE ,FALSE or NULL
- Anything compared with NULL results in a NULL value.
- A NULL in an expression evaluates to NULL (except concatenation)

E.g.

5 + NULL evaluates to NULL

‘PL/’ || NULL || ‘SQL’ evaluates to ‘PL/SQL’

| PL /SQL Datatypes | Operators |
|---|---|
| NUMBER | < , > |
| CHAR | =, != |
| DATE | <=, >= |
| BOOLEAN | |

# IF Statements

**‘IF’ Statements** are used to conditionally execute the statement or sequence of statements.

**‘IF’ Statement Syntax**

```
IF <condition> THEN  <sequence of statements >
    [ELSEIF <condition> THEN <sequence of statements> ]
    -- ELSEIFs may be repeated
    [ELSE <sequence of statements>]
END IF;
```

**QuickNotes - IF Statement**

1. <condition> must evaluate to a Boolean datatype (TRUE, FALSE, NULL)

2. If <condition> is TRUE, then the associated <sequence of statements> is executed; otherwise it is not

3. At most one <sequence of statements > gets executed

# IF Statements

```
DECLARE
 num_jobs    NUMBER(7);
BEGIN
    SELECT COUNT(*) INTO num_jobs FROM auditions
    WHERE actorid=&&actor_id AND called_back =‘YES’;
    IF num_jobs> 90 THEN
        UPDATE actor SET actor_rating = ‘ OSCAR time’
        WHERE actorid = &&actor_id;
    ELSE IF num_jobs> 75 THEN
                  UPDATE actor SET actor_rating = ‘ DAY time soaps’
                  WHERE actorid = &&actor_id;
    ELSE
        UPDATE actor SET actor_rating = ‘ Waiter’
         WHERE actorid = &&actor_id;
    END IF;
   COMMIT;
END;
```

# IF Statements

## The NULL Trap

**BLOCK 1**

```
        .
IF  a >= b   THEN
  do_this …..;
ELSE
   do_that….;
END IF;
```

**BLOCK 2**

```
        .
IF  a < b   THEN
  do_that …..;
ELSE
   do_this….;
END IF;
```

- Given any pair of non-NULL values for “a” and”b”, will Block 1 and Block 2 do the same thing?
- What if either “a” or”b” (or both) is NULL?

# Loop Statement Overview

**Loops** repeat a statement or sequence of statements multiple times.

**Four types of loops:**

1. Simple Loops.
2. Numeric FOR Loops.
3. While Loops.
4. Cursor FOR Loops.

# Loop Statements

**Simple Loops** repeat sequence of statements multiple times.

**Simple Loop Syntax**

```
Loop
  <Sequence of Statements>
END  LOOP ; -- sometimes called an 'infinite' loop
```

**Exit statements** exit any type of loop immediately

**Exit Syntax**

```
EXIT [WHEN <condition >]; -- 'infinite' loop insurance
```

# Loop Statements ……Example

```
DECLARE
   ctr   NUMBER(3) := 0;
BEGIN
   LOOP
     INSERT INTO LOG VALUES (ctr, ’ITERATION COMPLETE’);
     ctr := ctr +1;
     IF ctr = 1500 THEN EXIT;
     END IF;
  END LOOP;
END;
```

```
DECLARE
   ctr   NUMBER(3) := 0;
BEGIN
   LOOP
     UPDATE  TABLE 1 SET COMMIT = ‘UPDATES’ WHERE COUNT_COL = ctr;
     ctr := ctr +1;
      IF ctr = 1500 THEN EXIT;
      END IF;
   END LOOP;
END;
```

# Loop Statements

**Numeric FOR Loops** repeat sequence of statements fixed number of times.

## Numeric FOR Loop Syntax

FOR <index> IN [REVERSE ] <integer>..<integer> LOOP <sequence of statements>

**The Loop Index** takes on each value in range , one of a time , either in forward or reverse order.

**E.g.**

```
BEGIN
   FOR I IN 1..500 LOOP
      INSERT INTO temp(message)VALUES (‘I  will not sleep in class.’);
   END  LOOP;
END;
```

# Loop Statements

## QuickNotes - Index

1. It is implicitly of type NUMBER

2. It is only defined within the loop

3. Value may be referenced  in an expression, but a new value may not be assigned to it within the loop

**E.g.**

```
DECLARE
   my_index CHAR(20) := ‘Fettuccini Alfredo’;
BEGIN
   FOR my index IN REVERSE 21…30 LOOP  /* redeclare s my_index*/
      INSERT INTO temp(coll.)VALUES (my_index); /* insert the numbers 30 through 21*/
    END LOOP;
END;
```

```
FOR     i    I N   1…256   LOOP
        x  :=  x  +  i ;            -- legal
        i   :=  I   + 5;            -- illegal
END LOOP;
```

# Loop Statements

**WHILE Loops** repeat a sequence of statements until a specific condition is no longer TRUE.

**While Loop Syntax**

WHILE <condition > LOOP <sequence of statements > END LOOP;

**QuickNotes - WHILE Loops**

1. The term <condition> may be any legal PL/SQL condition (i.e. it must return a Boolean value of TRUE, FALSE or NULL)
2. The sequence of statements will be repeated as long as <condition> evaluates to TRUE

```
DECLARE
    ctr          NUMBER (3)   :=      0;
BEGIN
    WHILE ctr  < 500 LOOP
        INSERT INTO  temp(message) VALUES (‘Well,I might sleep just a little’);
        ctr := ctr +1 ;
     END LOOP;
END;
```

# “ GO TO ” Statement Overview

**“ GO TO “ Statements** jump to a different place in the PL/SQL block.

**“GO TO” Statements** have 2 parts

1. The GOTO statement itself.
2. A statement ‘label’

**“GO TO “ Statement Syntax**

```
  <<label_name >> X :=X+1 ;  - - statement label
    GOTO LABEL_NAME      - - JUMPS TO  x  := x +1
```

# “ GO TO ” Statements

NOT ALL GOTOs are Legal !

You can legally a GOTO  a statement that is either:

1.in the same sequence of statements as the GOTO STATEMENT

2. In the sequence of statements that encloses the GOTO statement (I.e. an outer block)

```
<<dinner>>
        x := x + 1 ;
        y := y  + 1;
IF  a >=  b   THEN
     b : = b + c;
      GOTO dinner;
END IF;
```

```
GOTO your_brothers;
IF  a > b   THEN
     b := b - a;
     <<your_brothers>>
     x := x - 1;
END IF;
```

# Other Uses for Statement Labels

**Labels** may label any statement

In addition to their use as targets for GOTO statements, labels may be used for :

1. Blocks
2. Loops

Labeling a block allows referencing of DECLARED objects that would otherwise not be visible because of Scoping rules.

**Syntax**

```
<< label_name>>
[  DECLARE
        --  declarations go here   ]
BEGIN
        --  executable statements go here
 [ EXCEPTION
       -- exception handlers go here  ]
 END  label_name ;  --  must include the label_name
```

# Other Uses for Statement Labels

**E.g.**

```
<< outer_block >>
DECLARE
      n       NUMBER;
BEGIN
      n   :=    5;
      /*   Start  a sub block   */
      DECLARE
           x    NUMBER   :=  10;
           n    CHAR (10)  :=  'Fifteen';
       BEGIN
           INSERT INTO TEMP VALUES (outer_block.n , x , n );
           COMMIT;
       END ;  /*   End of the sub block   */
 END      outer_block;
```

# Other Uses for Statement Labels

**Labeling a Block** allows you to reference a variable that might be hidden by a column name

**E.g.**

```
<< sample  >>
DECLARE
   deptno              NUMBER  := 20;
BEGIN
   UPDATE emp SET sal = sal * 1.1
   WHERE deptno = sample.deptno;
   COMMIT;
END sample;
```

# Other Uses for Statement Labels

**Labeling Loops** allows you to reference objects that would otherwise not be visible because of scoping rules

**E.g.**

```
<< compute_loop >>
For i IN  1…10  LOOP
     < statements    …. >
      DECLARE
            i     NUMBER   := 0 ;
      BEGIN
         INSERT INTO temp VALUES
              (i, compute_loop.i, 'COMPLETE' );
      END;
END LOOP compute_loop;  - must include loop name here
```

# Other Uses for Statement Labels

**Labeling EXITs** is a convenient way to specify exits from outer loops

**E.g.**

```
<< outer_loop >> WHILE    a > b    LOOP
   b := b + 1;
    << inner_loop >> WHILE    b > c    LOOP
        c := c  + 2 ;
        EXIT outer_loop WHEN    c > 200 ;
    END LOOP inner_loop;
END LOOP outer_loop;
```

# PL/SQL

## Cursors

# Cursor Overview

**Every SQL DML statement processed by PL/SQL has an associated CURSOR.**

**Two Types of CURSORS**

1. **EXPLICIT**

   Multiple row SELECT STATEMENTS

2. **IMPLICIT**

   All INSERT statements

   All UPDATE statements

   All DELETE statements

   Single row SELECT….INTO Statements

# Using Explicit Cursors

## *STEP 1 . Declare the cursor*

```
DECLARE
            CURSOR <cursor name> IS  <regular select statement> ;
```

## QuickNotes - CURSOR Declaration

1. The < regular select statement > must NOT include the INTO clause required in a single-row SELECT….INTO statement

2. Declared cursors are scoped just like variables

## Cursor Declaration Example

```
DECLARE
            X                      NUMBER (  7, 2  ) ;
            total                  NUMBER ( 5 )
            lower_sal_limit        CONSTANT NUMBER ( 4 )    :=  1200 ;
            CURSOR c1 IS SELECT ename FROM emp WHERE sal > lower_sal_limit ;
BEGIN ...
```

# Using Explicit Cursors

## ***STEP 2 . Open the cursor***

OPEN < cursor name > ;

## ***STEP 3 . Fetch data from the cursor***

FETCH < cursor name >   INTO  < var1 ,var2 > ;

**Quick Notes  - FETCH**

1. Retrieves one row of data from the cursor and stores it in the specified variables (similar to how a single-row select works)

2. There must be exactly one INTO  variable for each column selected by the SELECT statement

3. The first column gets assigned to var1 , the second to var2 , etc .

## ***STEP 4 . Close the cursor***

CLOSE < cursor name > ;

# Explicit Cursors Attributes

## %NOTFOUND

**E.g.**

```
LOOP
        FETCH my_cursor INTO my_ename , my_sal ;
        EXIT WHEN my_cursor%NOTFOUND ;
          --  process data here
END LOOP ;
```

## %FOUND

**E.g.**

```
FETCH my_cursor INTO my_ename ,my_sal ;
WHILE my_cursor%FOUND LOOP
     --  process data here
    FETCH my_cursor INTO my_ename ,my_sal ;
END LOOP ;
```

# Explicit Cursor Attributes

## %ROWCOUNT

**E.g.**

```
LOOP
        FETCH my_cursor INTO my_ename , my_sal ;
        EXIT WHEN (my_cursor%NOTFOUND)
                OR  (my_cursor%ROWCOUNT > 10) ;
        --     process data here
END LOOP
```

## %ISOPEN

**E.g.**

```
IF my_cursor%ISOPEN THEN
        FETCH my_cursor INTO my_ename , my_sal ;
ELSE
        OPEN my_cursor ;
END IF ;
```

# Using Explicit Cursors

**E.g.**

```
DECLARE
        sal_limit                      NUMBER ( 4 ) :=  0 ;
        my_ename                       emp.ename%TYPE ;
        my_sal                         emp.sal%TYPE ;
        CURSOR my_cursor IS SELECT ename , sal FROM emp WHERE sal > sal_limit ;
BEGIN
        sal_limit  := 1200 ;
        OPEN   my_cursor INTO my_ename , my_sal ;
        LOOP
             FETCH my_cursor INTO my_ename , my_sal  ;
             EXIT  WHEN my_cursor%NOTFOUND ;    -- nothing returned
             INSERT INTO new_table VALUES ( my_ename , my_sal ) ;
        END LOOP ;
        CLOSE my_cursor ;
        COMMIT ;
END ;
```

# Explicit Cursors -FOR Loops

**Cursor FOR Loops** specify a sequence of statements to be repeated once for each row that is returned by the cursor.

**Cursor FOR Loop Syntax**

```
        FOR <record _name>  IN  <cursor_name> LOOP
              --- statements to be repeated go here
        END LOOP;
```

**Numeric FOR Loop Similarities**

1. Specify a set of rows from a table by using the cursor’s name vs. specifying a set of integers (i.e. 1…10)

2. Index takes on the values of each row vs. index taking on integer values (i.e. 1 through 10)

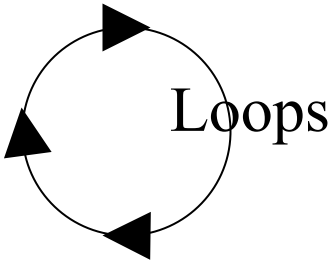
**Implicitly Declared <record_name>**

```
        record_name   cursor _name%ROWTYPE;
```

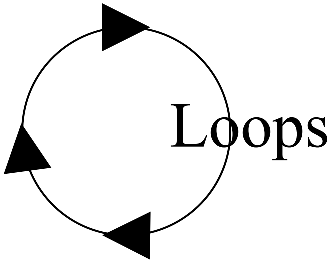
To reference an element of the record, use the record_name.column_name notation.

# Explicit Cursors -FOR Loops

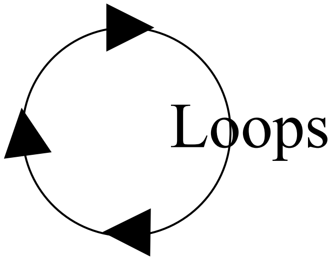
## Conceptual Cursor Loop  Model



When a cursor loop is initiated, an implicit OPEN cursor_name is executed.



For each row that satisfies the query associated with the cursor an, implicit FETCH is executed into the components of record_name.



When there are no more rows left to FETCH, an implicit CLOSE cursor_name is executed and the loop is exited.

# Explicit Cursors - FOR Loops

**E.g.**

```
DECLARE
    sal_limit                    NUMBER ( 4 )    :=   0 ;
    total_sal                     NUMBER (9,2 )  :=   0;
    CURSOR my_cursor IS SELECT ename , sal FROM  emp
    WHERE  sal > sal_limit ;
BEGIN
    sal_limit  := 1200 ;
    -- implicitly  OPEN done next
     FOR  cursor_row IN my_cursor LOOP
         -- an implicit fetch done here
         INSERT INTO new_table VALUES (cursor_row.ename ,cursor_row.sal ) ;
         total_sal := total_sal + cursor_row.sal;
      END LOOP ;   --an implicit close done here.
     COMMIT ;
END ;
```

# Implicit Cursors - FOR Loops

An Implicit Cursor is automatically associated with any SQL DML statement that does not have an explicit cursor associated with it.

**This includes :**

1. ALL INSERT statements
2. ALL UPDATE statements
3. ALL DELETE statements
4. ALL SELECT…INTO statements

**QuickNotes - Implicit Cursors**

1. Implicit cursor is called the "SQL" cursor --it stores information concerning the processing of the last SQL statement not associated with an explicit cursor.

2.OPEN, FETCH, AND CLOSE don't apply.

3. All cursor attributes apply.

# Implicit Cursors

## SQL %NOTFOUND

**E.g.**

```
UPDATE emp SET sal = sal * 10.0
WHERE ename  =“WARD” ;

IF SQL %NOTFOUND THEN
    -- WARD wasn’t found
    INSERT INTO emp (empno, ename ,sal)
    VALUES ( 1234,’WARD’ 99999 );
END IF ;
```

# Implicit Cursors

## SQL%ROWCOUNT

**E.g.**

```
DELETE FROM baseball_team
WHERE batting _avg. < .100;


IF SQL%ROWCOUNT > 5 THEN
   INSERT INTO temp(message)
   VALUES(‘Your team needs helps.’);
END IF;
```

# PL/SQL

## Exception Handling

# Exception Overview

- In PL/SQL error are called exceptions
- When an exception is raised, processing jumps to the exception handlers
- An exception handler is a sequence of statements to be processed when a certain exception occurs
- When an exception handler is complete processing of the block terminates

# Exception Overview

**Two Types of Exceptions**

1. PREDEFINED INTERNAL EXCEPTIONS

2. USER-DEFINED EXCEPTIONS

**PL/SQL's Exception Handlers**

**vs.**

**Conventional Error Handling**

# Predefined Internal Exceptions

**Any ORACLE error “raises” an exception automatically; some of the more common ones have names.**

TOO_MANY_ROWS ORA-(01427)

- a single row SELECT returned more than one row

NO_DATA_FOUND ORA-(01403)

- a single row SELECT returned no data

INVALID_CURSOR ORA-(01001)

- invalid cursor was specified

VALUE_ERROR ORA-(06502)

- arithmetic ,numeric, string , conversion,or constraint error occurred.

ZERO_DIVIDE ORA-(01476)

- attempted to divide by zero

DUP_VAL_ON_INDEX ORA-(00001)

- attempted to insert a duplicate value into a column that has a unique index specified.

# Exception Handlers

**Syntax**

```
WHEN <exception_name [OR <exception_name…]> then <sequence of statements>
                    OR
WHEN OTHERS THEN    --  if used , must be last handler <sequence of statements>
```

**E.g.**

```
DECLARE
    employee_num    emp.empno%TYPE;
BEGIN
    SELECT empno INTO employee_num FROM emp;
    WHERE ename = ‘BLAKE’;
    INSERT INTO temp VALUES(NULL, empno,Blake's employee_num’);
    DELETE FROM emp  WHERE ename =‘BLAKE’;
EXCEPTION
   WHEN TOO_MANY_ROWS OR NO_DATA_FOUND THEN
       ROLLBACK;
       INSERT INTO temp VALUES (NULL,’Blake not found, or more than one Blake’);
       COMMIT;

   WHEN  OTHERS THEN
      ROLLBACK;
END;
```

# User - Defined Exceptions

**User - defined Exceptions** must be defined and explicitly raised by the user.

**E.g.**

```
DECLARE
   x                       NUMBER;
   my_exception            EXCEPTION; -- a new object type.
```

**Raise your_exception;**

```
RAISE my_exception;
```

**Quick Notes -RAISE <exception_name>**

1. Once an exception is RAISED manually, it is treated exactly the same as if it were a predefined internal exception.

2. Declared exceptions are scoped just like variables.

3. A user-defined exception is checked for manually and then RAISED , if appropriate.

# User - Defined Exceptions

**E.g.**

```
DECLARE
    my_ename    emp.ename%TYPE :=‘BLAKE’;
    assigned_projects     NUMBER;
    too_few_projects   EXCEPTION
BEGIN                  ----  get no of projects assigned to BLAKE
     IF  assigned_project < 3 THEN
        RAISE   too_few_projects;
    END IF;
EXCEPTION      --begin the exception handlers
    WHEN too_few_projects THEN
        INSERT INTO temp
        VALUES(my_ename,assigned_projects,’LESS THAN 3 PROJECTS!’)
        COMMIT;
END;
```

# Exceptions Propagation

**Step# 1** The current block is searched for a handler. If not found, go to step 2.

**Step# 2** If an enclosing block is found, it is searched for it handler.

**Step# 3** Step #1 and#2 are repeated until either there are no more enclosing blocks, or a handler is found .

- If there are no more enclosing blocks, the exception is passed back to the calling environment (SQL *Plus,SQL *Forms, a precompiled program,etc.)
- If the handler is found ,it is executed .when done the block in which the handler was found is terminated, and control is passed to thee enclosing block (if one exists), or to environment (if there is no enclosing block)

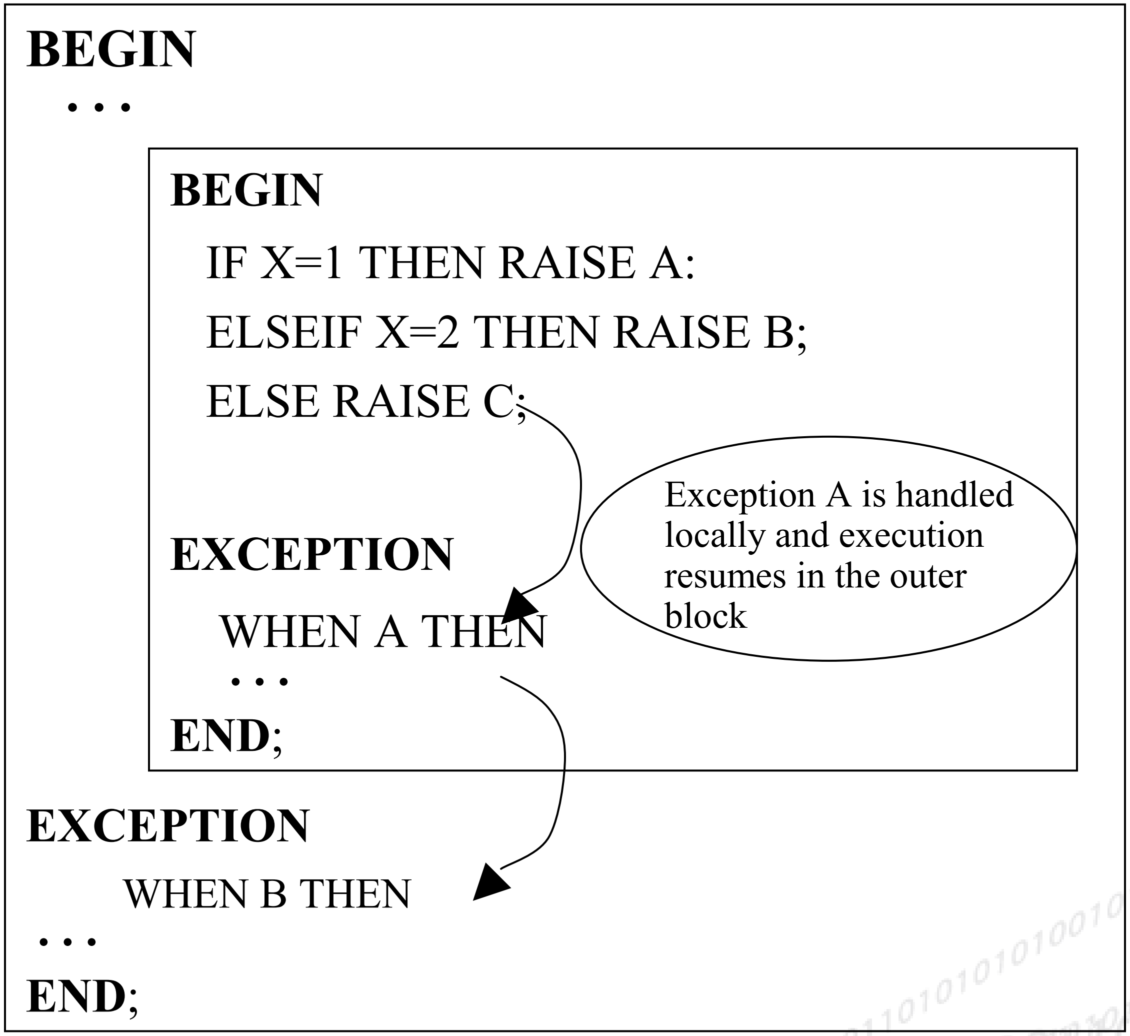
## Quick notes

1. Only one handler per block may be active at a time

2. If an exception is raised in a handler, the search for a handler for the new exception begins in the enclosing block of the current block

# Exceptions Propagation

Example 1

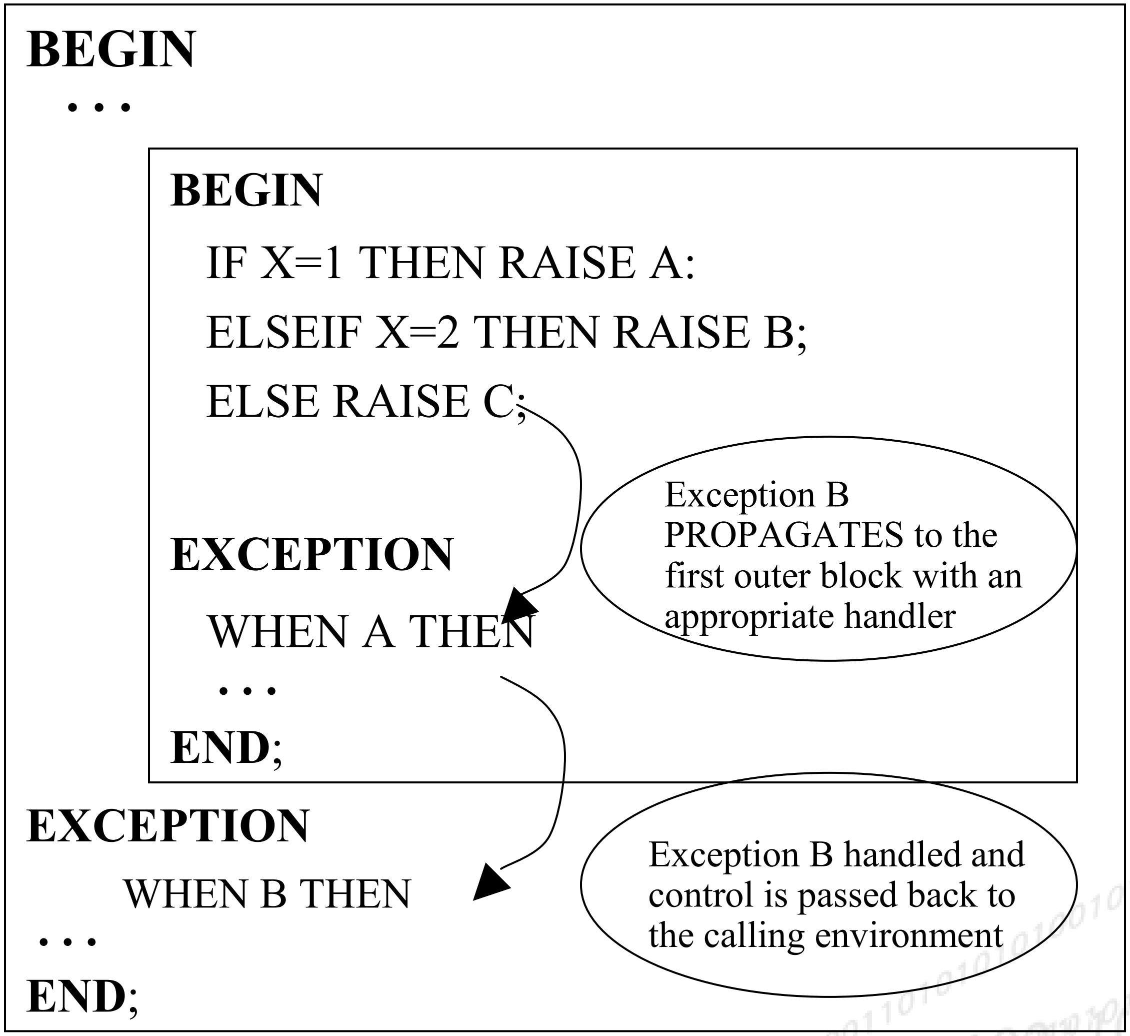


```
BEGIN
  . . .
        BEGIN
          IF X=1 THEN RAISE A:
          ELSEIF X=2 THEN RAISE B;
          ELSE RAISE C;

        EXCEPTION
           WHEN A THEN
           . . .
        END;
EXCEPTION
       WHEN B THEN
 . . .
END;
```

Exception A is handled locally and execution resumes in the outer block

# Exceptions Propagation

Example 2



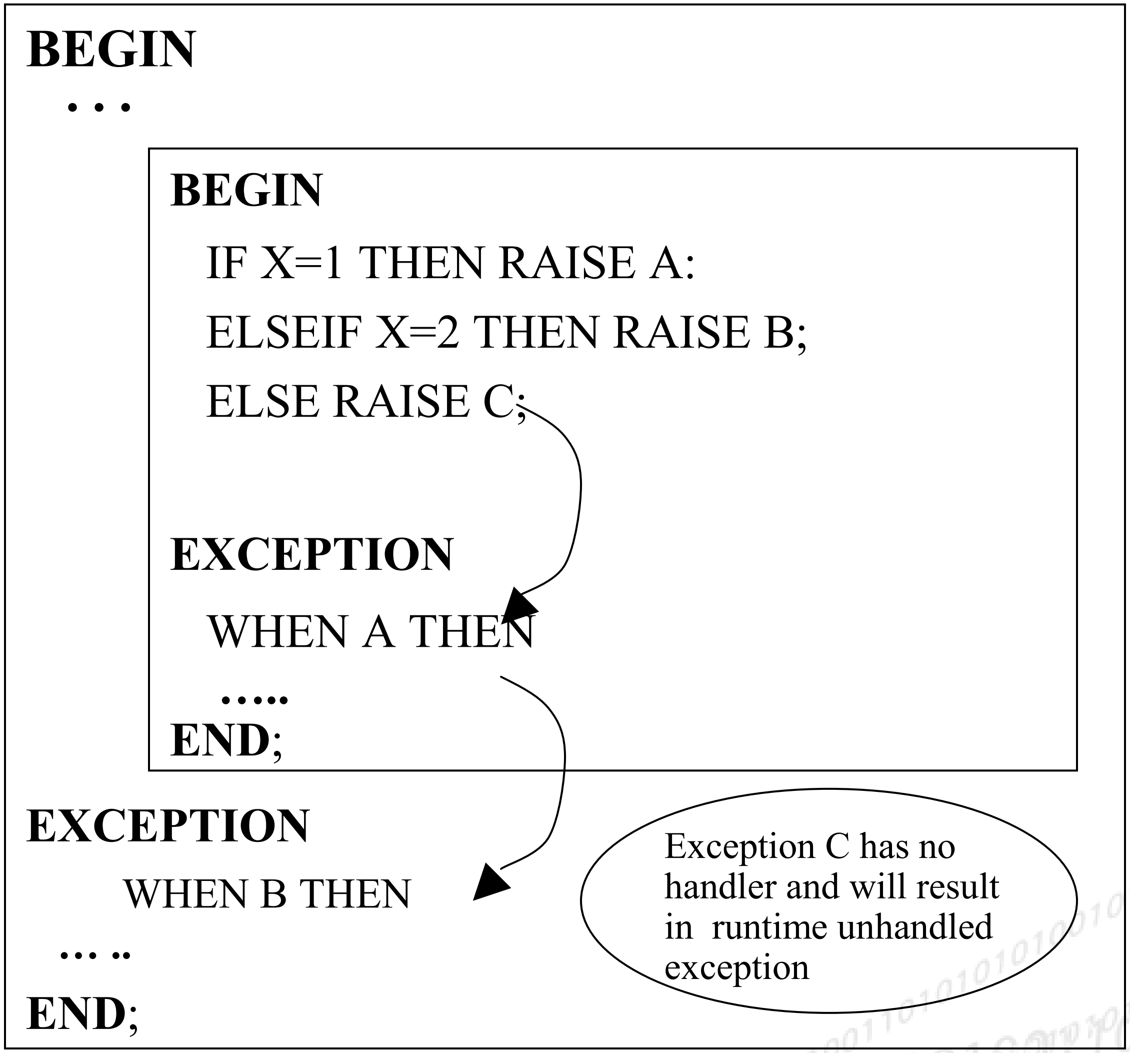
```
BEGIN
  . . .
      BEGIN
        IF X=1 THEN RAISE A:
        ELSEIF X=2 THEN RAISE B;
        ELSE RAISE C;

      EXCEPTION
        WHEN A THEN
          . . .
      END;
EXCEPTION
      WHEN B THEN
 . . .
END;
```

Exception B PROPAGATES to the first outer block with an appropriate handler

Exception B handled and control is passed back to the calling environment

# Exceptions Propagation

Example 3



```
BEGIN
  . . .
      BEGIN
        IF X=1 THEN RAISE A:
        ELSEIF X=2 THEN RAISE B;
        ELSE RAISE C;

      EXCEPTION
        WHEN A THEN
         …..
      END;
EXCEPTION
      WHEN B THEN
  … ..
END;
```

Exception C has no handler and will result in runtime unhandled exception

## Other uses of RAISE

By itself ,the **RAISE** statement simply re-raise the current exception (as if it were being propagated)

**Syntax**

```
RAISE;
```

**Quick Notes - RAISE**

1. RAISE may only be used in an exception handler

# Error Reporting Functions

## SQLCODE and SQLERRM

1. Provided information on the exception currently being handled.
2. Especially useful in the OTHERS handler.

## SQLCODE

1. Returns the ORACLE error number of the exception or 1 if it was user-defined exception

## SQLERRM

1. Return the ORACLE error message currently associated with the current value of SQLCODE
2. May also use any ORACLE error number as an argument.

## QuickNotes - Error Reporting

1. If no exception is active …

   SQLCODE = 0

   SQLERRM = “normal , successful completion”

2. SQLCODE and SQLERRM cannot be used within a SQL statement.

# Error Reporting Functions

**E.g.**

```
DECLARE
    sqlcode_val       NUMBER;
    sqlcode_val       CHAR(70);
BEGIN
…
EXCEPTION
  WHEN OTHERS THEN
      sqlcode _val := SQLCODE;  -- can't insert directly
      sqlerrm_val  := SQLERRM ;  - -- can't insert directly
      INSERT INTO temp VALUES(sqlcode_val,  NULL, sqlerrm_val);
END;
```

# PL/SQL

## Procedures & Packages

# Stored Procedures and Functions

- Collections of SQL and PL/SQL statements
- Stored in complied from in the database
- Can call other procedures
- Can be called from all client environments
- Procedures and function (including remote) are the same, except a function returns a values and a procedure does not.

## Uses for Procedures

- Define central well-known business functions.
  - Create  an order
  - Delete a customer
- Store batch job in the database
  - Weekly account rollups
- Encapsulate a transaction
  - Gather and process information from remote nodes
- Funnel activity on a table through a single path
  - All changes to employee salaries should change department budgets.

# Procedure Arguments

## Argument Modes

**IN** Data value comes in from the calling process and is not changed

**OUT** No data value comes in from the calling process; on normal exit ,value of argument is passed back to caller

**IN OUT** Data value comes in from the calling process, and another value is returned on normal exit

# Creating a Procedure

- **E.g.**

```
CREATE PROCEDURE
    fire_employee  (empid NUMBER)
AS
BEGIN
   …
   DELETE
     FROM emp
     WHERE empno=
          fire_employee.empid;
END
```

- **Tip**:Write each procedure in a text file, and save

# Creating a Function

**E.g.**

```
CREATE FUNCTION
      get_bal  (acc_no  NUMBER(4))
   RETURN       NUMBER(11,2);
IS
    acc_bal   NUMBER(11,2);
BEGIN
     SELECT   balance
         INTO    acc_bal
         FROM   accounts
         WHERE  account_id_no=acc_no;
  RETURN   (acc_bal);
END;
```

# Statements in procedures

O **Valid statements in a procedure or function**

- SQL DML or PL/SQL statements
- Calls to other procedures and functions stored in the database
- Calls to other procedures and functions in a remote database

O **Restricted statements**

- DDL
- Dynamic SQL
- In trigger, COMMIT, SAVEPOINT and ROLLBACK

# Executing a stored procedure

O **From within a PL/SQL block**

```
fire_employee  (empno);
scott.fire_employee (empno);
```

O **On a remote node**

```
scott.fire_employee@ny  (empno);
```

O **From SQL*DBA and some ORACLE tools**

```
EXECUTE  fire_employee  (1043)
```

O **Within an anonymous block (when EXECUTE not available)**

```
SQLPLUS>
BEGIN  FIRE_EMPLOYEE(1043); END;
```

O **From a precompiler application**

```
EXEC SQL
 FIRE_EMPLOYEE(:empno);
```

# Specifying procedure arguments

O **E.g.**

- CREATE PROCEDURE update_sal

```
(empno      NUMBER,
 bonus      NUMBER,
 sal_incr   NUMBER)  ….;
```

## O Positional Method

- List values in the order they are declared

```
update_sal   (7000,20,500);
```

## O Named Method

- List argument names and values in any order, using special syntax

```
update_sal
 (bonus => 20,
  sal_incr => 500,
  empno => 7000);
```

# Specifying procedure arguments

## O Combination method

- Use both positional and named methods
- Once you specify a named parameter, must use named method for all remaining parameters

```
update_sal                                   Legal
  (7000,sal_incr=>500,bonus=>20);


update_sal                                   Illegal
  (empno=>7000,
   sal_incr=>500,20);
```

# How procedures are entered into the database

O **PL/SQL Engine compiles the source code**

O **ORACLE stores a procedure in a database:**

- Object name
- Source code
- Parse tree
- Pseudo code(P-code)
- Syntax errors in dictionary table
- Dependency information

O **SQL in procedure not stored in parsed form**

- Uses shared and cached SQL
- Allows SQL to be optimized dynamically (without recompiling referencing procedures)

# PL/SQL Compilation Errors

O **Compile done by PL/SQL engine in RDBMS**

O **Error stored in the database**

O **To view errors:**

- Use SQL*DBA command SHOW ERRORS
- Select from errors views
- USER_ERRORS
- ALL_ERRORS
- DBA_ERRORS

# PL/SQL Compilation Errors Executing SHOW ERRORS

```
SQLDBA>create procedure test1 is begin test2;end;
DBA-00072: Warning: PROCEDURE TEST1
created with compilation errors.

SQLDBA>show errors
ERRORS FOR PROCEDURE TEST1:
LINE/COL ERROR
--------------------------------------------------------------------------------------
1/0         PL/SQL:  Compilation unit
analysis terminated
1/33        PL/SQL-00219:'test2' is not defined'
2 rows selected
```

# PL/SQL Compilation Errors

## Fields in ERRORS views

O **NAME**:name of the object

O **TYPE**: one of the following:

- PROCEDURE
- FUNCTION
- PACKAGE
- PACKAGE BODY

O **LINE**:line number where error occurs

O **TEXT**:text of error

# USER-DEFINED System Errors

- **Any procedure can raise an error and return a user defined error message and error number**

- **Error number range is -20000 to -20999**

- **Range always reserved for user defined errors**

- **Oracle does not check if user defined error numbers are used uniquely**

- **Raise error within PL/SQL block with procedure**

  raise_application_error

  (error_number,'text of the message')

- **Full pathname of procedure may be needed in early releases**

  sys.standard_utilities.

  raise_application_error

# USER-DEFINED System Errors

## Example

```
CREATE PROCEDURE
    fire_employee (empid NUMBER)
AS
BEGIN
  IF empid <=0 THEN
     raise_application_error (-20100,’Employee number must be> 0’);
 ELSE
     DELETE
        FROM  emp
        WHERE  EMPNO =EMPID;
 END IF;
END;

SQLDBA> EXECUTE FIRE_EMPLOYEE(-1);
ORA=-20100: Employee number must be >0
```

# Dependencies and Procedures

O **A procedure is dependent on:**

- every database object to which it refers (direct dependency)
  procedures, functions, packages, tables, views, synonyms, sequences

- the database objects those objects depend on (indirect dependency)

O **Two types of dependencies**

local: objects are on the same node

remote: objects are on separate nodes

O **ORACLE automatically checks dependencies**

# Recompilation of Dependent procedures

O **When an object changes, its dependent objects are marked for recompilation**

O **Any change to definition of referenced object implies new version of reference object**

O **Dependent objects must be recompiled if referenced object changes**

O **Recompilation of dependent objects takes place automatically at runtime**

O **Reasons recompilation could fail**

- Changing parameter list in called procedure
- Changing definition of or removing referenced column from referenced table
- Dropping referenced table

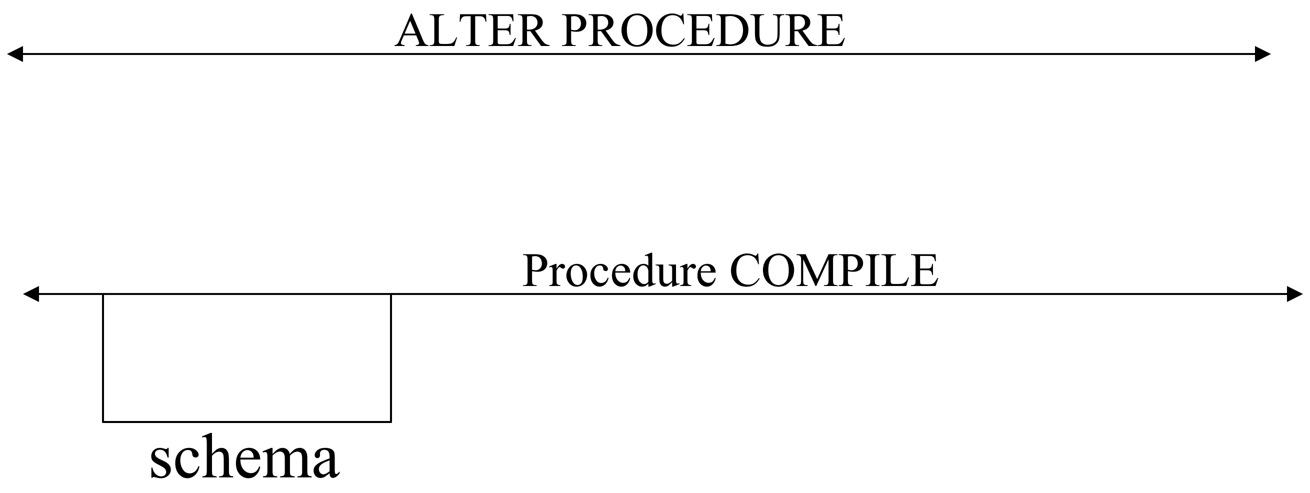
# Recompilation

O **Procedure/function can be recompiled be either**

- RDBMS automatically, when next accessed (only if marked for recompilation)
- Manually by the user, using ALTER PROCEDURE command

O **If recompilation fails, error information is put to error table**

# Manual Recompilation



ALTER PROCEDURE

Procedure COMPILE

schema

O **Example**

ALTER PROCEDURE

add_department COMPILE

# Changing a Procedure

O **To modify a procedure, replace it:**

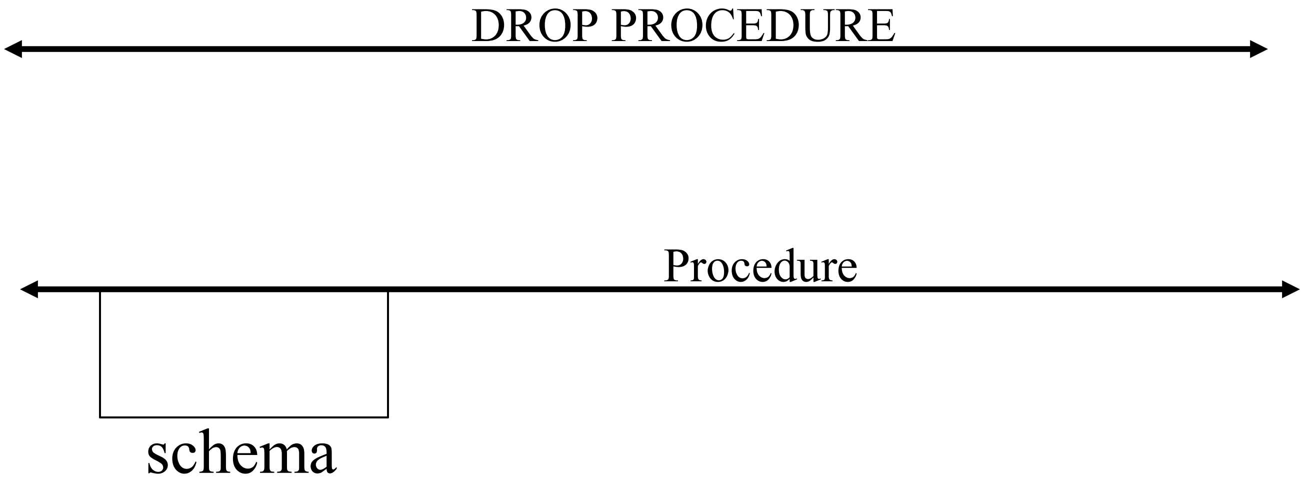
```
CREATE OR REPLACE PROCEDURE
   fire_employee AS . . . END;
```

O **OR REPLACE option:**

- Recreates the procedure even if it already exists
- Retains existing grants (need not reissue)
- Creates procedure even if there are syntax errors
- Marks dependent objects for recompilation
- Users executing old procedure finish that call: next invocation gets new procedure
- Facilitates development (one step)

O **CREATE without OR REPLACE on existing procedure fails**

# Dropping a Procedure



**Example**

DROP PROCEDURE fire_employee;

**Marks dependent objects for recompilation**

# Privileges for Procedures

- **Example**

```
GRANT EXECUTE
      ON scott.hire_fire
      TO mkennedy
       WITH GRANT OPTION;
```

- **Procedure executes under the authority of owner, not user executing procedure**

- **User of procedure need not have access to objects inside procedure**

- **Can only GRANT privileges on an entire package, not a procedure, function, or variable defined in the package**

# Privileges for Procedures

PROCEDURE system privileges apply to procedures, functions and packages

| To do this | Need either | And |
|---|---|---|
| **CREATE** | CREATE PROCEDURE or CREATE ANY PROCEDURE system privilege | Owner must have access to all objects referenced in the procedure |
| **ALTER** | Own the procedure or ALTER ANY PROCEDURE system privilege | |
| **DROP** | Own the procedure or DROP ANY PROCEDURE system privilege | |

# Privileges for Procedures

| To do this | Need either | And |
|---|---|---|
| Execute a procedure or access a package construct | Own the procedure or be granted EXECUTE PRIVILEGE or EXECUTE ANY PROCEDURE system privilege | Procedure owner must be explicitly granted access to all database objects in the procedure(not through roles) |

# Benefits of Procedures

O **Security**

- Executes under security domain of procedure's owner
- Provides controlled indirect access to database objects to non-privileged users

O **Integrity**

- Defines allowed operations on data
- Ensures related actions are performed together

O **Performance**

- Reduces number of calls to thedatabase
- Decreases network traffic
- Pre-parses PL/SQL statements

# Benefits of Procedures

O **Memory savings**

- Takes advantages of shared SQL
- Requires only one copy of the code for multiple users

O **Productivity**

- Avoids redundant code for common procedures in multiple applications
- Reduces coding errors: no redundant code written

# Benefits of Procedures

O **Maintainability**

- Enables system wide changes with one update
- Makes testing easier: duplicate testing not needed
- Dependency tracked by ORACLE

O **High availability**

- Allows changing procedured on-line while users execute previous version

# Package

O **A database object that groups related package constructs**

  ✓ Procedures

  ✓ functions

  ✓ cursor definitions

  ✓ variables and constants

  ✓ exception definitions

O **Package comprises**

  ➢ **Specification**

  ➢ **Body**

# Parts of a Package

O **Package specification**

- Declares (specifies) package constructs, including names and parameters publicly available procedures and functions

O **Package body**

- May declare additional, private package constructs that are not publicly available
- Defines all package constructs (public and private)
- May be replaced without affecting package specification (Breaks dependency chain)
- Each session has own version of state

# Public and Private Package Constructs

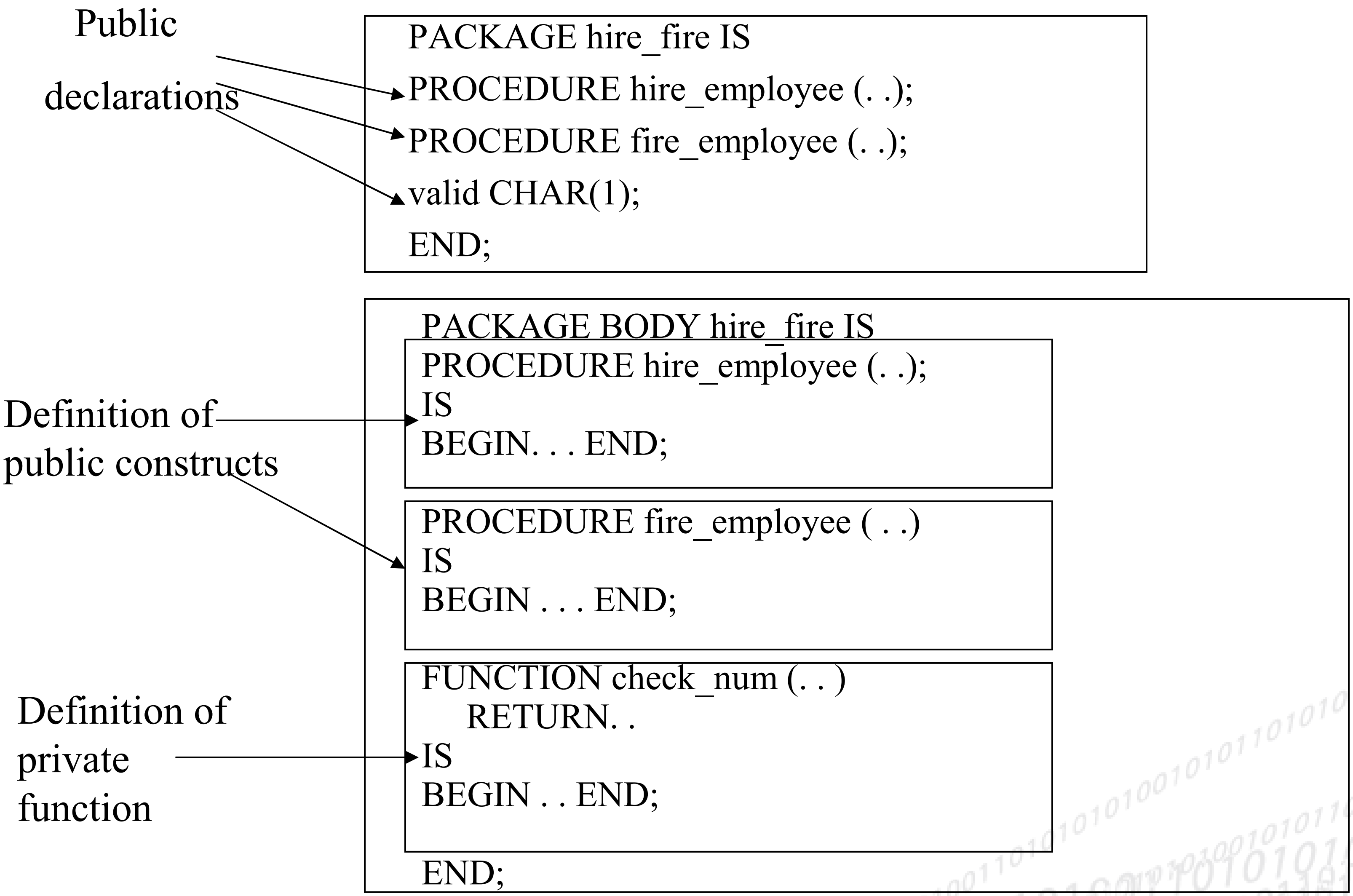
O **Public package constructs**

- Declared in the package specification
- Available to anyone who can run the package

O **Private package constructs**

- Only declared in the package body
- Available to other package constructs within the package body
- Allows hiding procedures from programmers referencing the public constructs

# Public and Private Package Constructs



Public declarations

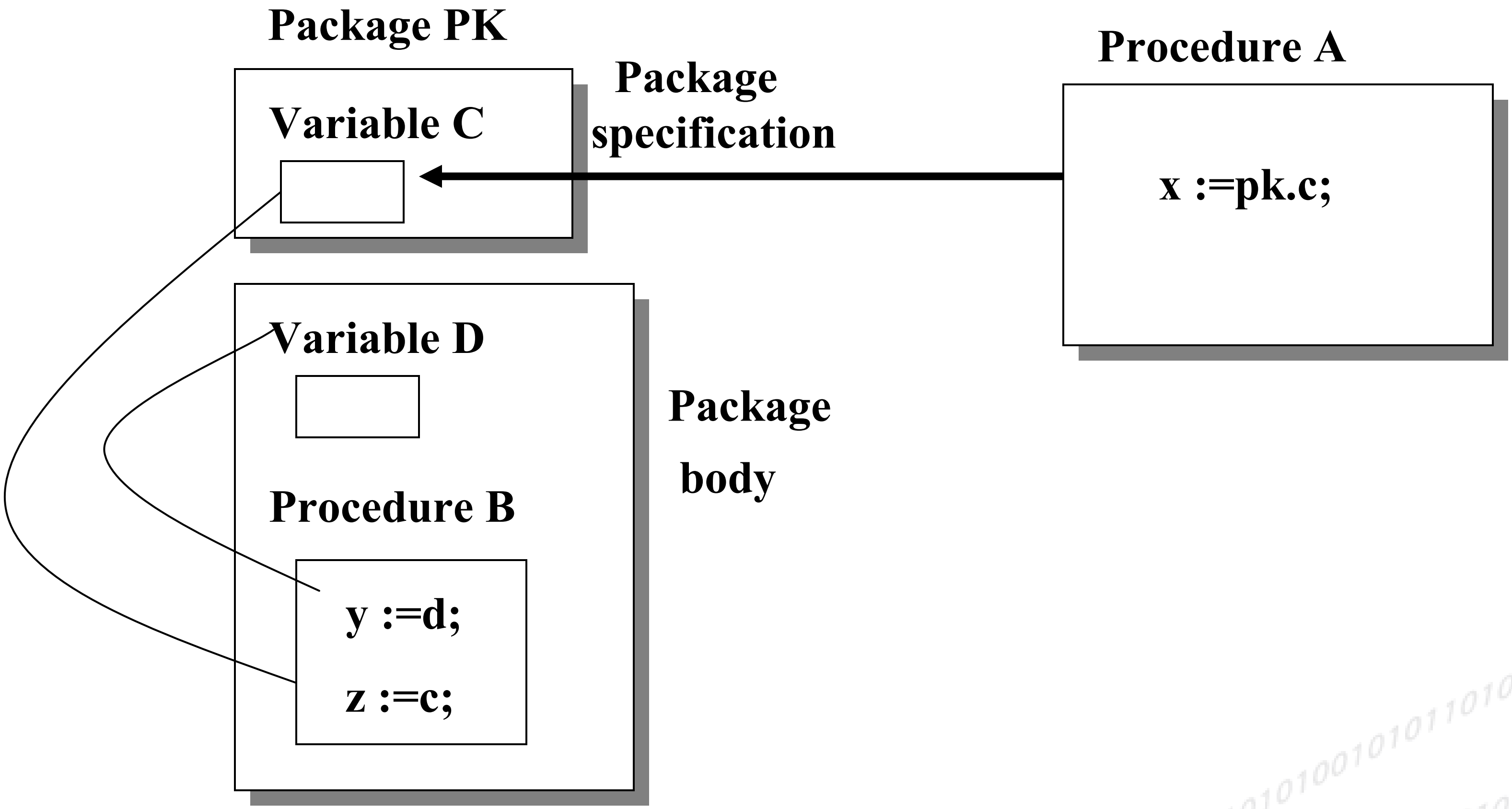
```
PACKAGE hire_fire IS
PROCEDURE hire_employee (. .);
PROCEDURE fire_employee (. .);
valid CHAR(1);
END;
```

Definition of public constructs

Definition of private function

```
PACKAGE BODY hire_fire IS
PROCEDURE hire_employee (. .);
IS
BEGIN. . . END;

PROCEDURE fire_employee ( . .)
IS
BEGIN . . . END;

FUNCTION check_num (. . )
    RETURN. .
IS
BEGIN . . END;

END;
```

# Public and Private Package Constructs



# Uses of Packages

O **Group related constructs**

O **Declare globally accessible variables**

O **Declare variables with persistent state**

O **Organize development activity**

- Define modules, collections of procedures known to on team

O **Minimize name conflicts within a schema**

- Personnel.audit ≠ inventory.audit

O **Simplify security**

- GRANT EXECUTE on entire package

O **limit recompilation**

- Change body of procedure or function without changing specification

# Creating a Package Specification

## Example

```
/* Package specification declaring procedures and variables for
hiring and firing employees */
CREATE PACKAGE hire_fire
AS
   /*Declare procedures */
   PROCEDURE hire_employee
       (empno     NUMBER,   ename     CHAR,
         mgr        NUMBER,   sal          NUMBER,
         comm      NUMBER,   deptno    NUMBER);
    PROCEDURE fire_employee (empid  NUMBER);
    /*Global variable declaration*/
    valid CHAR(1);
END hire_fire;
```

# Creating a package Body

## Example

```
/*package body containing procedures to hire and ire employee */
CREATE PACKAGE BODY hire_fire
AS
/*Procedure to hire a employee*/
PROCEDURE hire_employee
        (empno   NUMBER,   ename    CHAR,
          mgr      NUMBER,    sal          NUMBER,
          comm    NUMBER,    deptno   NUMBER);
IS
BEGIN
   /*VALID is declared in the package definition*/
     valid :=check_sum(empno);
     /*IF valid empno then add user*/
     IF valid =‘T’ THEN
          INSERT  INTO EMP VALUES
               (empno,ename,mgr,sal,comm,deptno);
     END IF;
END;
```

(continued)

# Creating a package body

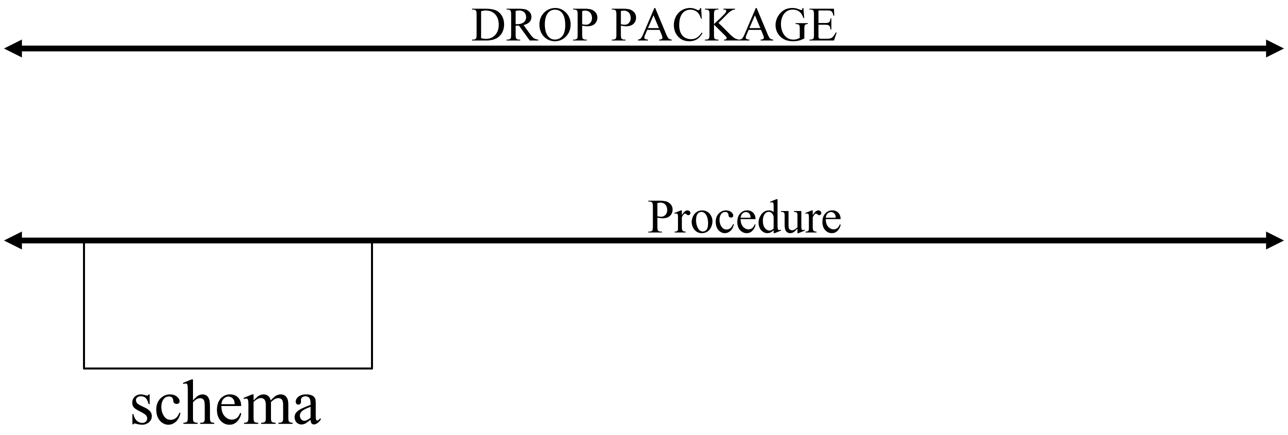
## Example(continued)

```
/* Procedure to fire an employee number */
PROCEDURE fire_employee  (empid  NUMBER)
 IS
 BEGIN
   DELETE  FROM  emp
        WHERE   empno  =empid;
 END;
/*function to check that the employee number>0.Local to the package */
FUNCTION check_sum (empno NUMBER)
   RETURN  CHAR(1)   IS
   answer       CHAR(1);
BEGIN
    answer :=‘T’;
    IF empno <0 THEN
      answer := ‘F’;
    END  IF;
    RETURN  (answer);
END;
END hire_fire;  /*End of package*/
```

# Accessing package constructs

```
PROCEDURE employee_admin
/* The action to perform and the employee ID have been entered previously*/
   IF action =“HIRE”THEN
     scott.hire_fire.hire_employee
        ( employee,ename,mgr,sal,comm,deptno);
     IF scott.hire_fire.valid =‘T’ THEN
     /*sports_club is another package that handles membership to the company
        sports club*/
        sports_club.add  (employee)
     END IF;
   ELSIF  action =“FIRE” THEN
        scott.hire_fire.fire_employee
            (employee);
        sports_club.remove  (employee);
   END IF;
```

# Dropping a Package



DROP PACKAGE

Procedure

schema

**Example**

DROP PACKAGE hire_fire;

# Benefit Of Package

Performance

- Reduces disk I/O for subsequent calls

  - First call to package loads whole package into memory

Persistence state

- Retain values of package constructs for an entire session

Security

- Access to the package allows access to public constructs in the package only.

- No need to issue GRANT for every procedure in package.

# Benefit Of Package

O **Productivity**

- •Stores related procedures and function together
- •Easier to manger changing the specification or definition of a construct.
- •Reduces cascading dependencies
  - •Dependencies are at package ,not procedure level
  - •Can change package body with changing or affecting package specification

# PL/SQL

## Triggers

# Triggers

O Definition

O Creating Triggers

O Restrictions on Triggers

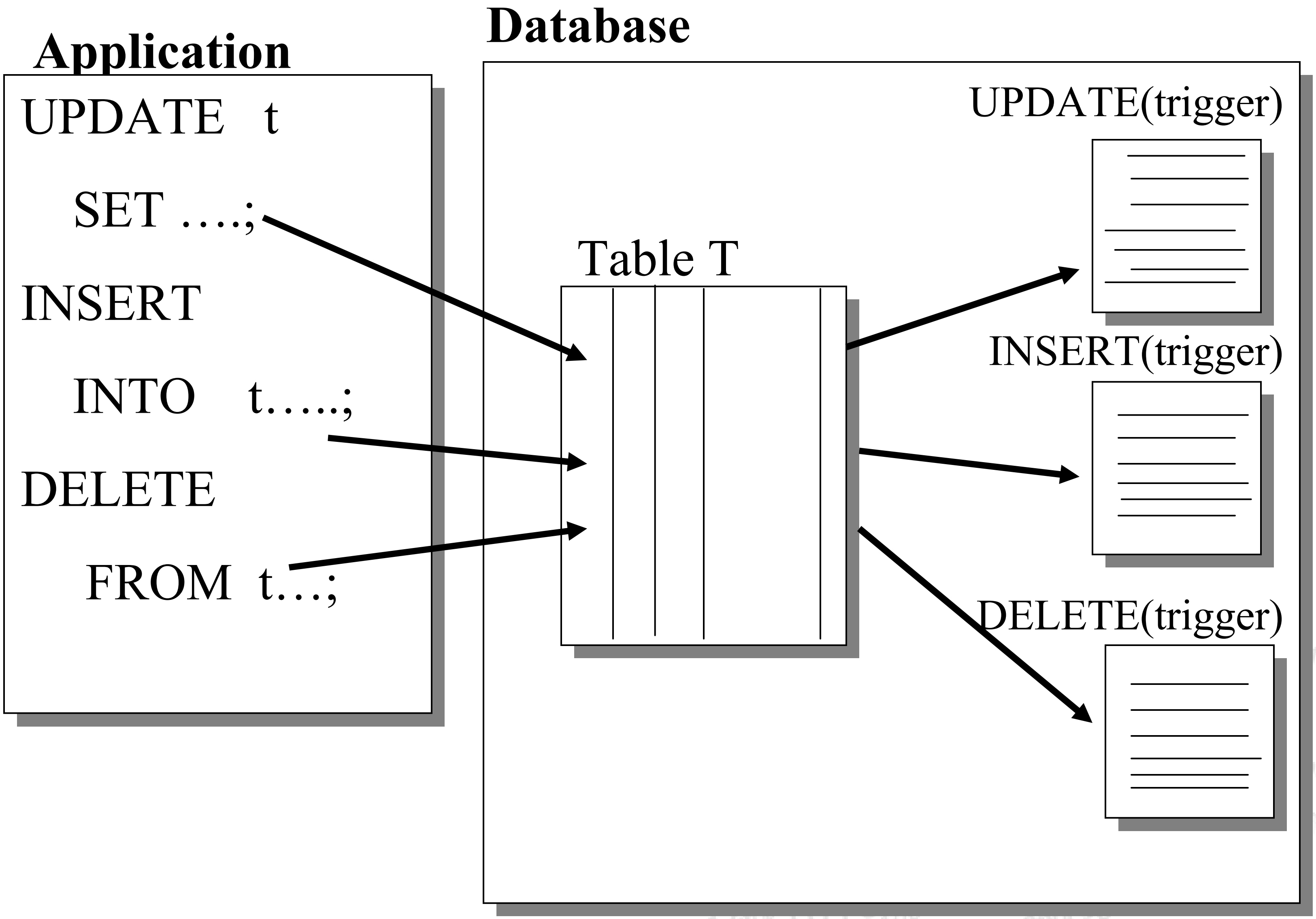
O Dropping and recompiling

O Privileges for creating

O Applications

O Benefits

# Trigger Definition



# What is a Trigger

O  A user-defined PL/SQL block associated with a specific  table, and implicitly fired (executed) when a triggering statement is issued against the table

O  Made up of parts

- Triggering event (INSERT/UPDATE/DELETE)
- Trigger type (BEFORE/AFTER, per statement or per row)
- Trigger restriction (optional)
    - *  WHEN clause
- Trigger action
    - *  PL/SQL BLOCK

O  Not the same as a SQL * Forms trigger

# E.g. of a Trigger - *Keeping salary in range for a job*

```
CREATE TRIGGER scott.salary_check
   BEFORE INSERT OR UPDATE sal, job ON scott.emp
   FOR EACH ROW
   WHEN (NEW .job <> ‘PRESIDENT’)
DECLARE
   minsal     NUMBER;
   maxsal  NUMBER;
BEGIN        /* get min and max salaries for the employee’s job from the SAL_GUIDE*/
   SELECT minsal,maxsal INTO minsal,maxsal FROM sal_guide
   WHERE job = :NEW.job;
   /* If salary below min or above max,generate an error*/
   IF (:NEW.sal < minsal.OR :NEW.sal > maxsal)
     THEN raise_application_error ( -20500,’salary’ || :NEW.sal|| ‘out of range for job’||
                                             :NEW.job|| ‘for employee’|| :NEW.ENAME);
   ENDIF;
END; /* End of Trigger*/
```

# Triggers and Stored procedures

## Similarities

- Made up of PL/SQL and SQL statements
- Use shared SQL areas
- Cannot be Changed (must be dropped and recreated )
- Dependencies tracked automatically by ORACLE

# Triggers and Stored procedures

## Differences

O Triggers invoked implicitly ;procedures invoked explicitly

O Triggers and procedures crated with different SQL statements

O No CREATE OR REPLACE TRIGGER statement

O Triggers are associated with tables

O COMMIT ,ROLLBACK,SAVEPOINT not allowed in Triggers (nor in procedures called by Triggers)

# Triggers vs. SQL*Forms Triggers

**Database trigger**

- Fires while statement executes
- Fires independently of and in addition to SQL *From Triggers
- Fired by SQL statement from any tool or application
- Can prevent completion of SQL statement
- Fire as each statement is executed

**SQL * Form trigger**

- Associated with particular form
- Only executes when from is run
- Can fire after each field is entered
- Pre/Post INSERT/UPDATE /DELETE Triggers execute when COMMIT key is pressed

# Types of Triggers

O **Type of a trigger determines**

- The time when the trigger fires
  - BEFORE trigger:

    before the triggering action
  - AFTER trigger:

    after the triggering action
- The item the trigger fires on
  - Row trigger: once for each row affected by the triggering statement
  - Statement trigger: once for the triggering statement, regardless of the number rows affected

O **Each table have up to 12 Triggers in all:**

INSERT/UPDATE/DELETE)

BEFORE/AFTER)

STATEMENT/ROW)

# Types of Triggers

## How to use each type

O **BEFORE statement trigger**

- To initialize global variables used in Triggers
- To prevent update before it occurs

O **BEFORE row trigger**

- To compute derived fields in the new row
- To prevent updates before they occur

O **AFTER row trigger**

- For auditing (by value,by row)
- Used by ORACLE snapshot mechanism

O **AFTER statement trigger**

- For auditing

# Trigger - Firing Sequence

O INSERT,UPDATE or DELETE is applied to table statement to execute

O Execute BEFORE statement trigger

O For each row affected by the SQL statement:

- Execute BEFORE row trigger
- Change row and perform integrity constraint checking
- Execute AFTER row trigger

O Execute AFTER statement trigger

O Complete deferred integrity constraint checking

O Returns to application

# Expressions in Triggers

## Referring to values in row Triggers

O To refer to the old and new values of a column in row Triggers, use the :OLD and :NEW prefixes:

- IF :NEW.sal < :OLD.sal. THEN

O Notes:

- Values available in row Triggers only
- New and old both available for UPDATE
- The old value in an INSERT is NULL
- The new value in a DELETE is NULL
- BEFORE row trigger can assign values to :NEW if it is not set by UPDATE SET clause or INSERT VALUES list
- Can replace :NEW and :OLD with other correlation names if desired
- Colon dropped in when clauses

# Expressions in Triggers

## Conditional Predicates

- If a trigger can fire on more than one type of DML operation  use pre defined PL/SQL boolean variables to determine which caused the trigger to fire:

```
IF INSERTING . . .
IF UPDATING  . . .
IF DELETING  . . .
```

- To detect which column is being updated:

```
IF UPDATING (‘columnname’)
```

# Expressions in Triggers

```
CREATE TRIGGER total_salary
    AFTER DELETE OR INSERT OR UPDATE
    OF deptno,sal ON emp
    FOR EACH ROW
BEGIN
    IF (DELETING) OR (UPDATING AND: OLD deptno <> :NEW deptno.)
           THEN UPDATE dept
              SET total_sal = total_sal -:OLD.sal
            WHERE deptno. = :OLD.deptno;
    END IF;
END;
```

# Restrictions on Triggers

O Maximum number of 12 Triggers for a table

- Up to three (INSERT/UPDATE/DELETE) Triggers of each type

O Prohibited statements in Triggers:

- ROLLBACK
- COMMIT
- SAVEPOINT

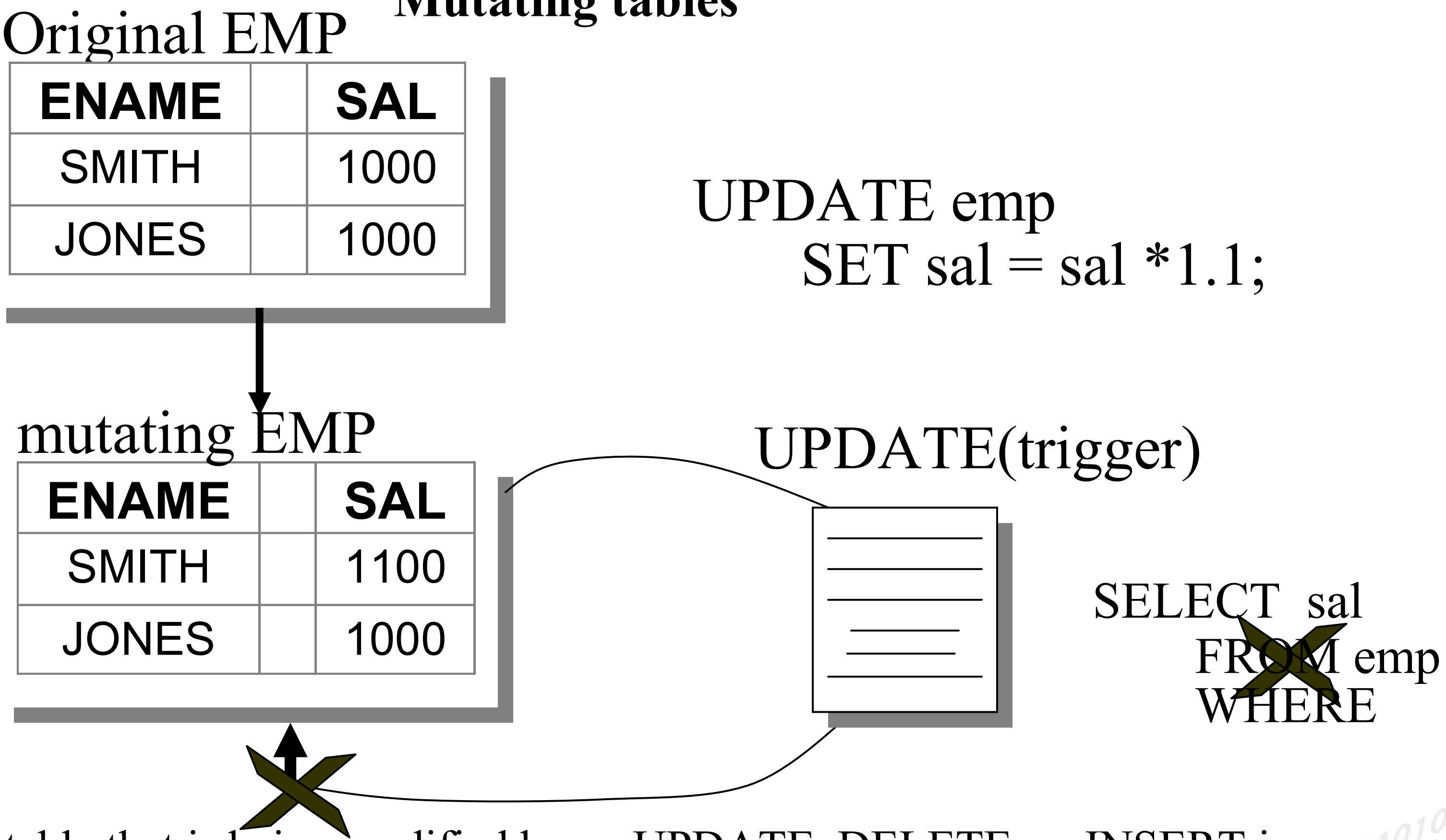
N.B. Also applies to procedures called by Triggers(including remote procedures

O Cascading trigger limit

- The action of a trigger may cause another trigger to fire and so on (called “cascading Triggers”)
- Limit the depth of cascading with an INIT.ORA parameter

# Restrictions on Triggers

## Mutating tables



Original EMP

| ENAME | | SAL |
|---|---|---|
| SMITH | | 1000 |
| JONES | | 1000 |

```
UPDATE emp
      SET sal = sal *1.1;
```

mutating EMP

| ENAME | | SAL |
|---|---|---|
| SMITH | | 1100 |
| JONES | | 1000 |

UPDATE(trigger)

```
SELECT  sal
      FROM emp
      WHERE
```

- A table that is being modified by an UPDATE, DELETE, or INSERT in a single user statement
- A trigger cannot SELECT from or change a mutating table (except current row, using :NEW and :OLD)

# Restrictions on Triggers

Changes to updated/inserted values

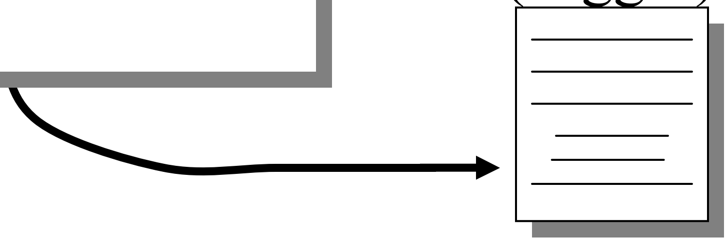
| EMPNO | | DEPTNO |
|---|---|---|
| 0450 | | 20 |
| 0407 | | 10 |

```
UPDATE emp
       SET deptno=10
             WHERE empno =0405;
```

mutating EMP

| EMPNO | | DEPTNO |
|---|---|---|
| 0450 | | 10 |
| 0407 | | 10 |

UPDATE (trigger)



~~GRATE TRIGGER bad~~
~~BEGIN~~
~~NEW.deptno:=30~~
~~END;~~

A trigger cannot change values explicitly referenced in the UPDATE statement SET clause or INSERT statement

# Enabling and disabling Triggers

O Possible states for a trigger

- Enabled
  - Executes its triggered action if both:

    an appropriate statement is issued.

    trigger WHEN clause evaluates to TRUE(if present).
- Disabled
  - •Does not execute its triggered action

O Triggers are automatically enabled on creation

# Enabling and disabling Triggers

## Reasons for disabling the trigger

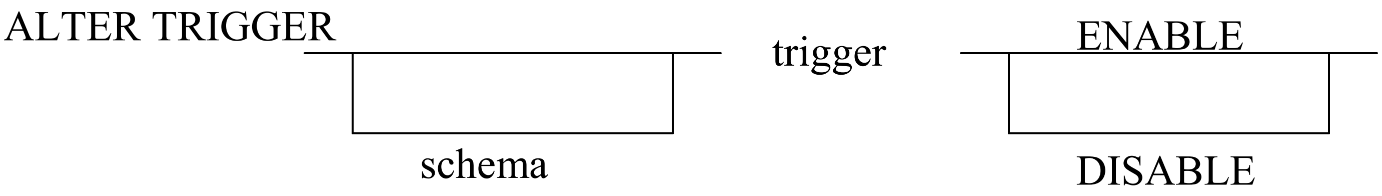
O Have to load a large amount of data, and want to proceed quickly without firing Triggers

- Example: SQL*Loader using direct path automatically disables Triggers

O Want to INSERT, UPDATE or DELETE on a table whose trigger references a database object that is not available

# Enabling and disabling Triggers

## With ALTER TRIGGER



ALTER TRIGGER

schema

trigger

ENABLE

DISABLE

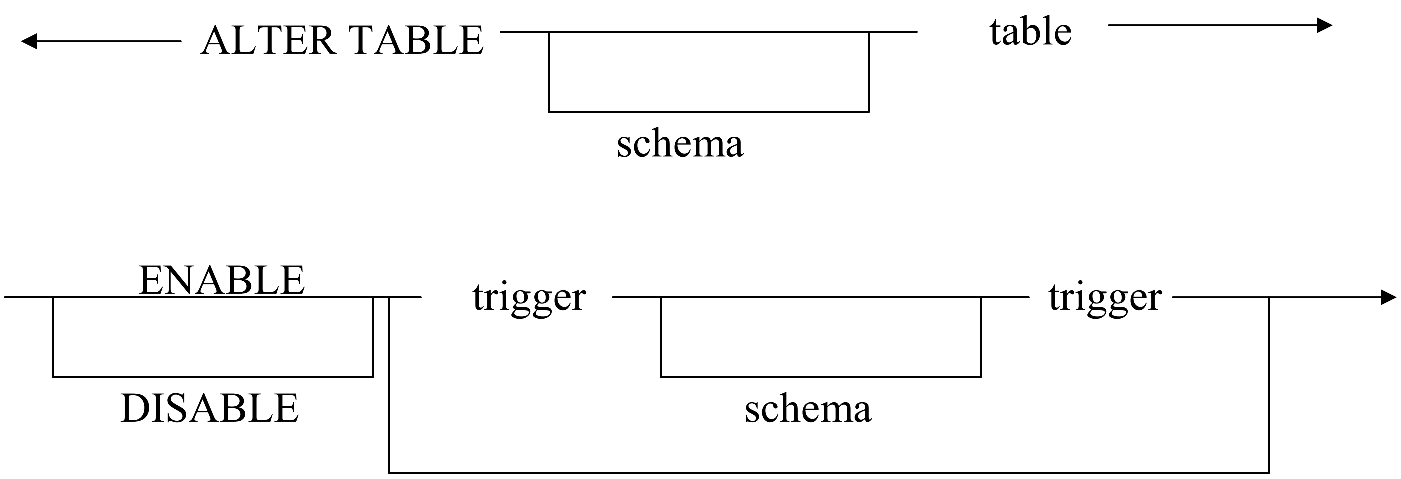
Examples

ALTER TRIGGER reorder DISABLE;

ALTER TRIGGER reorder ENABLE;

# Enabling and disabling Triggers

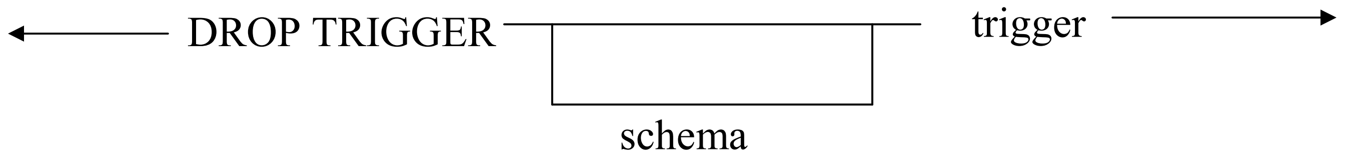
## With ALTER TABLE



- Examples

```
ALTER TABLE INVENTORY
   DISABLE TRIGGER REORDER;
ALTER TABLE INVENTORY
   ENABLE TRIGGER REORDER;
```

# Dropping Triggers



DROP TRIGGER

schema

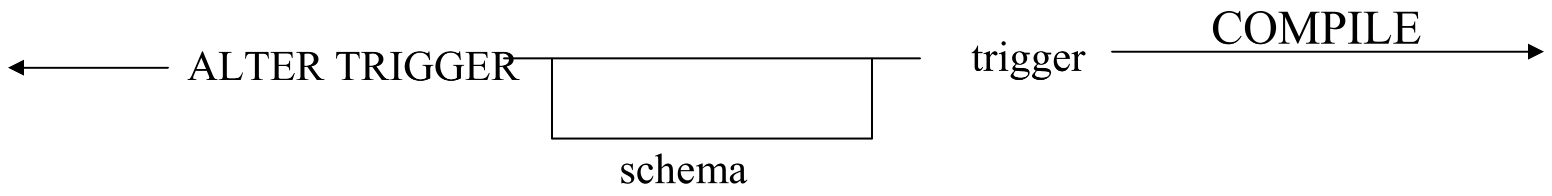
trigger

O Example

DROP TRIGGER reorder;

O Triggers cannot be altered; they must be dropped and recreated

# Recompiling a trigger



O Manually recompile to resolve dependencies (same as procedures)

O Example

```
ALTER TRIGGER reorder
    COMPILE;
```

# Applications of Triggers

- Maintaining derived fields
- Implementing complex security rules
- Enforcing complex business rules
- Performing value-based auditing
- Making implied changes
- Maintaining table replication

# Examples of using Triggers

## Deriving column values

- O Derive column values for a row based upon a value provided by an INSERT or UPDATE statement.
- O Must use BEFORE ROW trigger
  - Value must be derived first so the INSERT or UPDATE statement can use it.
  - Trigger must fire for each row affected.
- O Cannot assign to new values set by triggering INSERT or UPDATE.

# Examples of using Triggers

## Deriving column values

| Emp no | Ename | Uppername | Soundexname | Job |
|---|---|---|---|---|
| 7329 | Smith | Smith | S530 | Clerk |
| 7499 | Allen | Allen | A450 | Salesman |
| 7566 | Jones | Jones | J520 | Manager |

```
CREATE TRIGGER upper_soundex
    BEFORE INSERT OR UPDATE OF
          ename,
          uppername,
          soundexname
          ON emp ;
    FOR EACH ROW
    BEGIN
        :NEW.uppername := UPPER  (:NEW.ename);
        :NEW.soundexname := :SOUNDEX(:NEW.ename);
    END;
```

# Examples of using Triggers

## Complex Security Authorization

O Allows more complex security checking than provided by ORACLE

- Examples
  - Check for time of day,day of week
  - Check for terminal being used
  - Permit updates of certain amounts  by specific users

# Examples of using Triggers

## Complex Security Authorization

```
CREATE TRIGGER emp_permit_changes
   BEFORE INSERT OR DELETE OR UPDATE ON emp
DECLARE dummy INTEGER;
BEGIN
   IF(TO_CHAR (sysdate,’DY’) IN(‘SAT,’SUN’))
      THEN
       raise_application_error(-20504,’cannot change emp table during weekend’);
   END IF;
   SELECT  COUNT(* ) INTO dummy
      FROM company_holidays
      WHERE day = TRUNC(sysdate);
   IF dummy>0 THEN
       raise_application_error(-20505,’cannot change emp table during holiday’);
   END IF;
   IF (TO_NUMBER(sysdate,’HH24’)
      NOT  BETWEEN 8 AND 18) THEN
      raise_application_error (-20506,cannot change emp table in of_hours’);
   END IF;
   END;
```

# Examples of using Triggers

## Enforcing complex Business Rules

- Complex check constraints that are not definable using declarative constraints
- Can be used to maintain data integrity across a distributed database

(declarative integrity cannot span distributed database)

- Note: simple constraints are best handled by declarative constraints features

# Examples of using Triggers
## Enforcing complex Business Rules

```
CREATE TRIGGER increase_chk
    BEFORE UPDATING OF sal ON emp
   FOR EACH ROW
 WHEN (NEW.sal <OLD.sal OR
                   NEW.sal >1.1 * OLD. Sal)
  BEGIN
       raise_application_error(-20502,
        'may not decreased salary.
         Increase must be <10%')'
END;
```

# Examples of using Triggers

## Enforcing complex Business Rules

```
CREATE TRIGGER scott.salary_check
   BEFORE INSERT OR UPDATE OR UPDATE OF sal,
   ON scott.emp
   FOR EACH ROW
   WHEN (NEW.job <>’PRESIDENT’)
  DECLARE          minsal   NUMBER;
                   maxsal  NUMBER;
  BEGIN
      SELECT minsal,maxsal
        INTO  minsal,maxsal
        FROM    sal_guide
        WHERE job= :NEW.job ;
      IF (:NEW.sal <minsal OR
          :NEW.sal > maxsal)
      THEN raise_application_error ( -20503,’salary’ || :NEW.sal|| ‘out of range for
                              job’||:NEW.job|| ‘for employee’|| :NEW.ENAME);
       END IF;
END;
```

# Examples of using Triggers

## Value based auditing

O Auditing that cannot be accomplished with standard RDBMS auditing features

OAllows

- Exclusively DML auditing
- Per row auditing
- Storage of update details

OHowever Triggers cannot audit:

- DDL
- SELECTs
- Logons

# Examples of using Triggers

## Value based auditing

```
CREATE TRIGGER audit_employee
AFTER INSERT OR DELETE OR UPDATE ON emp
FOR EACH ROW
BEGIN
  IF auditpackage.reason IS NULL THEN
       raise_application_error(-20501,’MUST specify reason for  update before performing
                                    update;use auditpackage.set_reason()”’);
  END IF;
  INSERT INTO audit_employee VALUES
       (  :OLD.ssn,
           :OLD.name;
           :OLD.classification,
           :OLD.sal,
          :NEW.ssn;
          :NEW.name,
          :NEW.classification,
          :NEW.sal,
           auditpackage.reason,user,sysdate);
END;
```

# Examples of using Triggers

## Implied data changes

PENDING_ORDERS

| PART NO | ORD QTY | ORD DATE |
|---|---|---|
| 00234 | 15 | 15-JAN-92 |
| 00342 | 25 | 15-JAN-92 |

INVENTORY

| PART_NO | ON_HAND | REORD_PT | REORD_QTY |
|---|---|---|---|
| 00234 | 34 | 30 | 15 |
| 00342 | 52 | 50 | 25 |

O Transparently perform an action when a triggering is executed

O Example: Inventory check generates a new order

# Examples of using Triggers

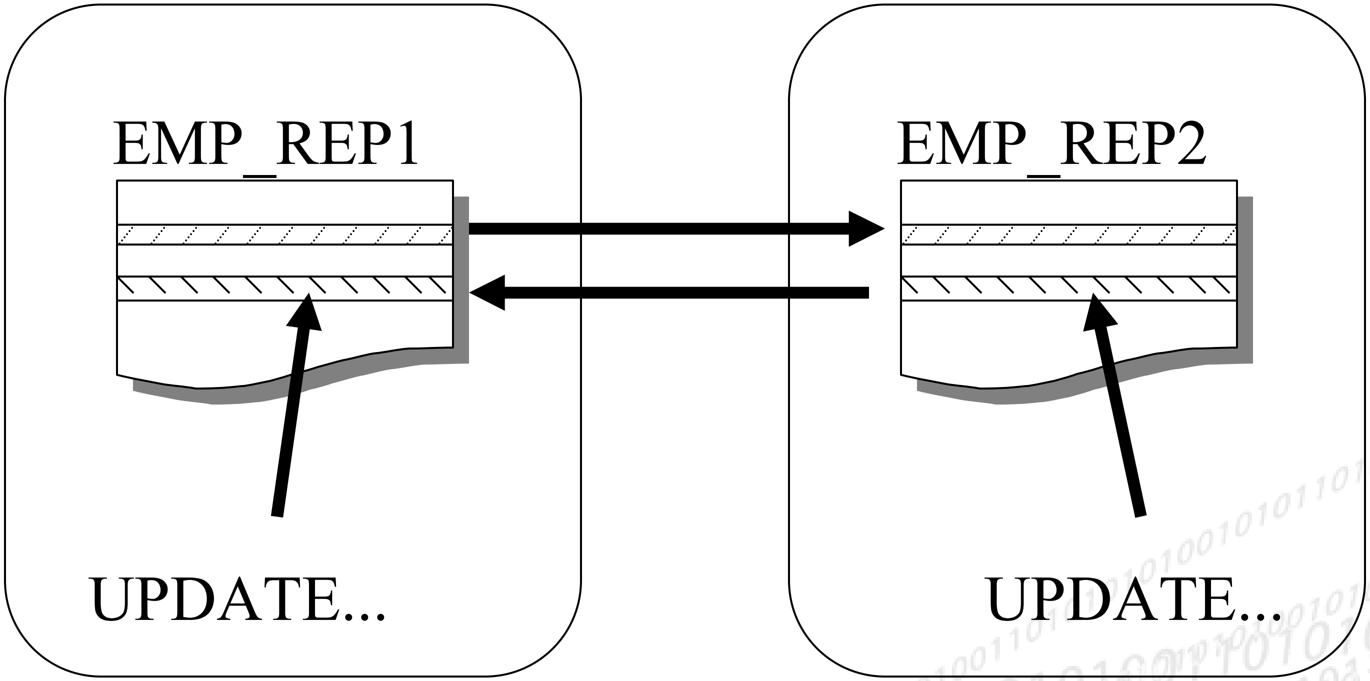
## Implied data changes

```
CREATE TRIGGER inv_check
AFTER UPDATE of on_hand
ON inventory
FOR EACH ROW
WHEN (NEW.ON_HAND <= NEW.reord_pt)
DECLARE x NUMBER;
BEGIN
     SELECT COUNT(*)
     INTO x
     FROM pending_orders
     WHERE pending_orders.part_no = :NEW.part_no;

     IF x = 0 THEN
        INSERT INTO pending_orders
        VALUES
         (:NEW.part_no,
           :NEW.reord_qty,
           SYSDATE);
      END IF;
END;
```

# Examples of using Triggers
## Synchronous Table Replication

Link identical tables(replicas) on different nodes so when replica is altered, changes are synchronously reflected in other replicas

Replicas must contain a flag field that is set when they are updated to stop trigger cascading



# Benefits of Triggers

O Security

- Allows sophisticated security checking
- Enables customized auditing mechanism to be built

O Integrity

- Ensures related operations on data are performed together
- Can be used to enforce complex business rules

# Benefits of Triggers

O Performance

- Reduces number of calls to the RDBMS
- Decreases network traffic

O Memory savings

- Takes advantage of shared SQL
- Requires only one copy of the code for multiple users

O Productivity

- Requires only a single copy of the code be written and maintained(not multiple copies in client applications)

# Thank you