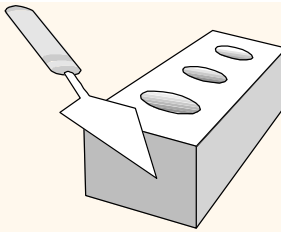
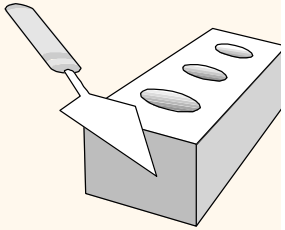


PL/SQL

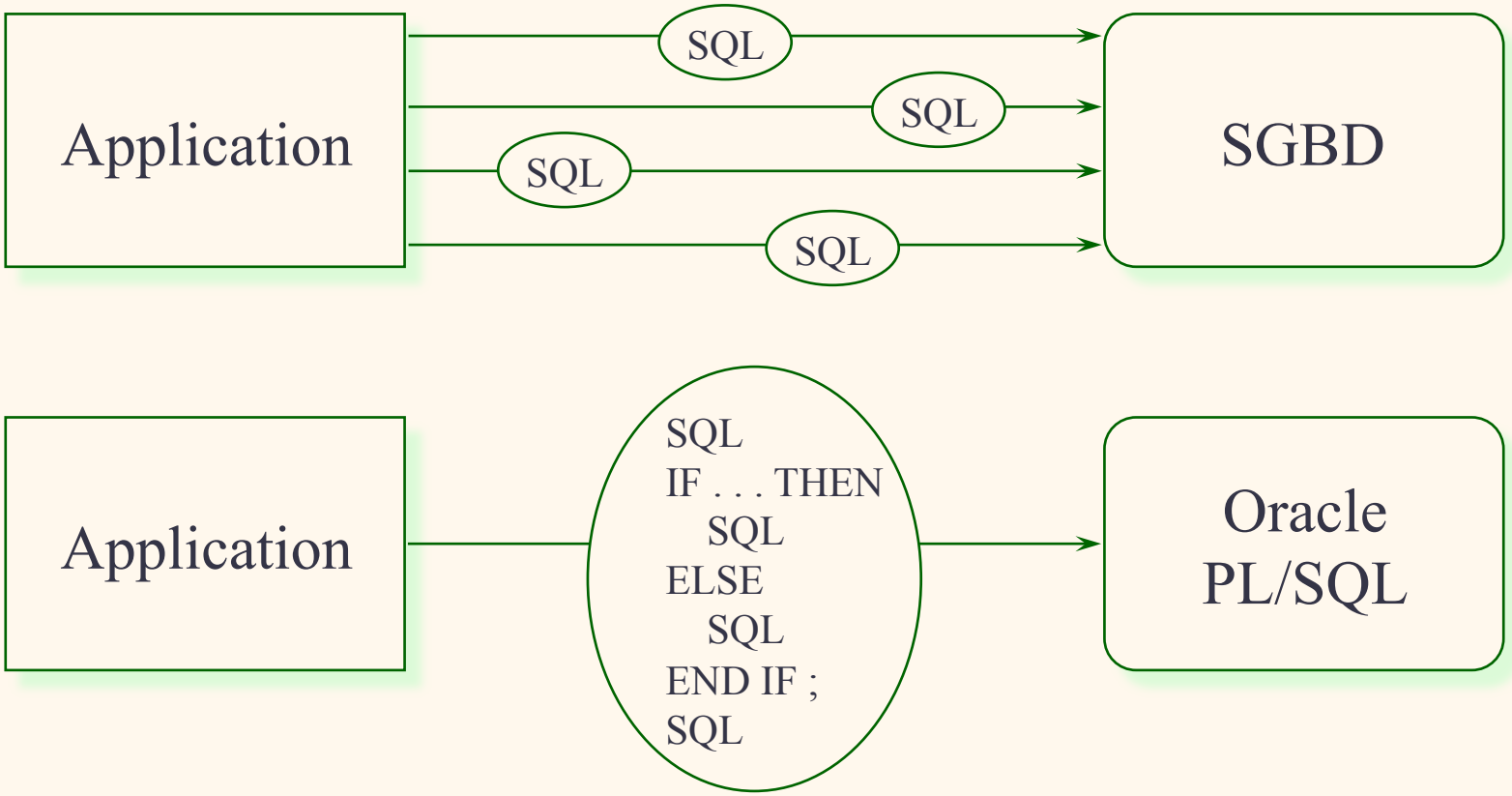
Raja CHIKY

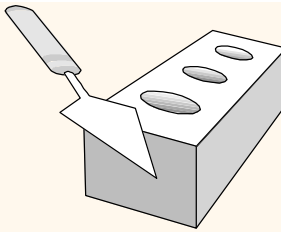
http://infolab.stanford.edu/~ullman/fcdb/oracle/or-plsql.html

Oracle® Database PL/SQL User's Guide and Reference
10*g* Release 2 (10.2)



PL/SQL

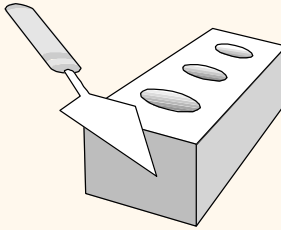

Application
SQL
SQL
SQL
SQL
SGBD
Application
SQL
IF . . . THEN
SQL
ELSE
SQL
END IF ;
SQL
Oracle
PL/SQL



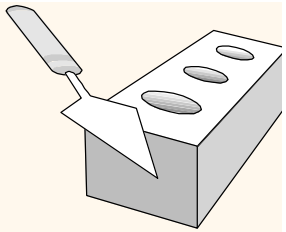
PL/SQL

- PL/SQL is a procedural language which allows
  - the declaration of variables
  - Development of complex database applications
    - Control structures (conditional, iterations …)
    - Procedural elements (procedures, functions, …)
  - Main goals of PL/SQL
    - Increase expressivity of SQL
    - Process the results of a query one tuple at a time (cursors)
    - Optimize the execution of a set of SQL commands
    - Reuse the programs' code

PL/SQL



- PL/SQL groups SQL queries in one block which is sent to the server
- PL/SQL improves the performances (less communications through the network)
- It is a portable language: it can function on any platform supporting Oracle Server
- Allows to create libraries of reusable code



PL/SQL

Anonymous Blocks

Database Triggers

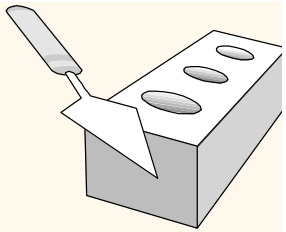
```
DECLARE
   o o o
BEGIN
   o o o
EXCEPTION
   o o o
END;
```

Stored Procedures

Packages

Stored Functions

MODULAR development of programs



Blocks

A PL/SQL block is composed of 3 sections

[block-header] *(optional)*

1 DECLARE (*optional*)
- variables, constants, cursors, user-exceptions

BEGIN (*required*)

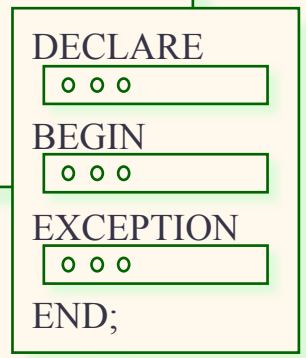
2
- order SQL
- order PL/SQL

EXCEPTION (*optional*)

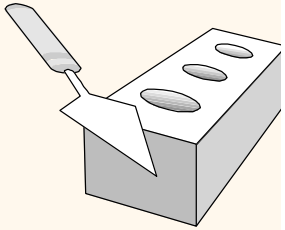
3
- Actions to carry out when an exception is raised or when an error takes place

END; (*required*)

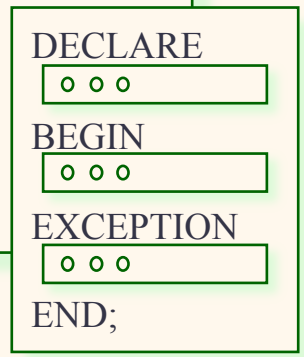
/



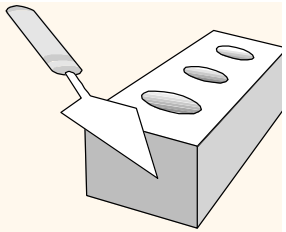
Blocks



```
DECLARE
        variable_v      VARCHAR2(5)
BEGIN
        SELECT          colonne_c
                INTO    variable_v
                FROM    table_t ;
EXCEPTION
        WHEN exception_e THEN
        . . .
END;
/
```



Blocks



Anonymous

```
[DECLARE]


BEGIN
. . .

[EXCEPTION]
. . .
END;
/
```

Procedure

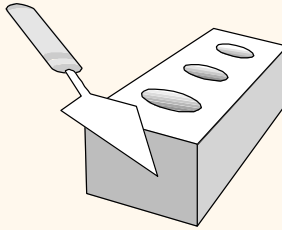
```
PROCEDURE <nom>
IS

BEGIN
. . .

[EXCEPTION]
. . .
END;
/
```

Fonction

```
FUNCTION <nom>
RETURN <type>
IS
BEGIN
. . .
RETURN <valeur> ;
[EXCEPTION]
. . .
END;
/
```

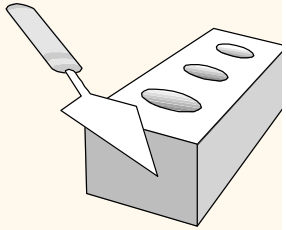
Comments :

```
-- comments on a line
/* comments on
several lines*/
```



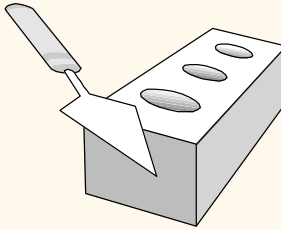
Structure of a PL/SQL block

- Block-header: indicates wether the block is a procedure, a function, a package (module)
  - A block without header is an anonymous block
- SQL commands usables in a PL/SQL block
  - All SQL/DML commands (SELECT, INSERT, UPDATE, …)
  - SQL/DDL commands cannot be used in PL/SQL blocks (create table, create view, create index, drop table, …)



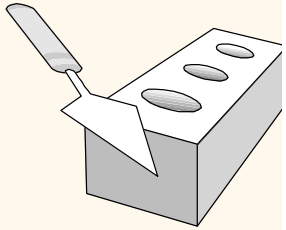
PL/SQL Variables and Types

- Information is transmitted between a PL/SQL program and the database through *variables*. Every variable has a specific type that can be
  - One of the types used by SQL for database columns
  - A generic type used in PL/SQL such as NUMBER
  - Declared to be the same as the type of some database column



PL/SQL Variables and Types

- The most commonly used generic type is NUMBER. Variables of type NUMBER can hold either an integer or a real number. The most commonly used character string type is VARCHAR2(*n*), where *n* is the maximum length of the string in bytes. This length is required, and there is no default.



Variables and constants

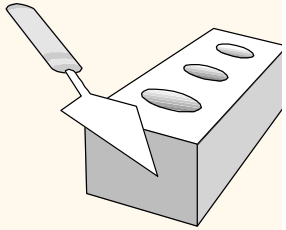
- Syntax :

  <name> [CONSTANT] <type> [NOT NULL]
  [:= | DEFAULT <expression>] ;

- Example :

```
DECLARE
        dateEmprunt_v     DATE ;
        noDept_v                  NUMBER(2) NOT NULL := 10 ;
        lieu_v                 VARCHAR2(13) := 'Paris' ;
        taux_c              CONSTANT NUMBER := 20 ;
```

- Note: constants and variable NOT NULL must be immediately affected



Variables and constants

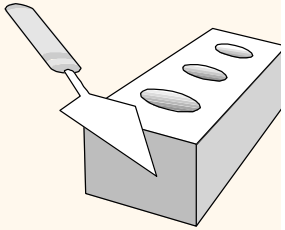
- Syntax :

  <variable_name> := <expression>

  or

  SELECT . . .
  INTO < variable_name >
  FROM . . . WHERE . . .

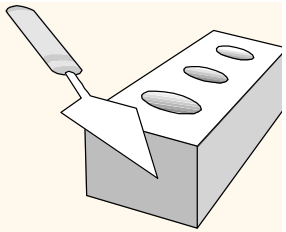
- Initialization of the variables:
  - Operator of assignment ':='
    nom_v := 'Toto' ;
    dateEmprunt_v := '31-DEC-2004' ;
  - DEFAULT
    chemin_g VARCHAR2(125) DEFAULT 'C:\progra~1\monAppli' ;
  - NOT NULL
    salaire_v NUMBER(4) NOT NULL := 0 ;



Variables and constants

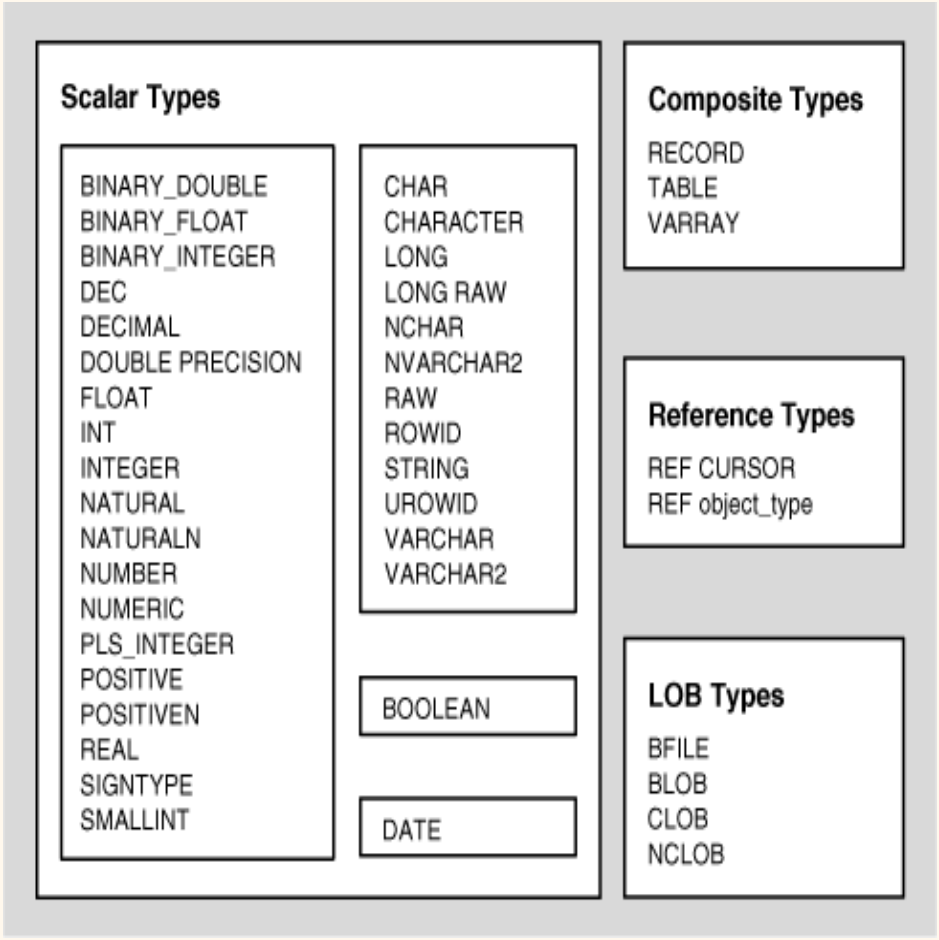
- Assignment:

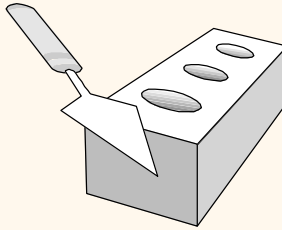
  < variable_name > := <expression>
- <expression> can be :
  - a constant
  - a variable
  - an operation with constants and variables
- operators of calculation:
  - arithmetic operators: + - * / **
  - operator of concatenation: ||
  - logical operators :
    - comparisons : < > = <= >= <>
    - connectors : AND OR NOT



Scalar types

- CHAR [(<*size_max*>)]
  character strings fixed length (max 32767)
- VARCHAR2 (<*size_max*>)
  character strings variable length (max 32767)
- NUMBER [(<p>, <s>)]
  Number having precision p and scale s
- DATE
- BOOLEAN
  three possible values: TRUE, FALSE and NULL





User Defined Type: record

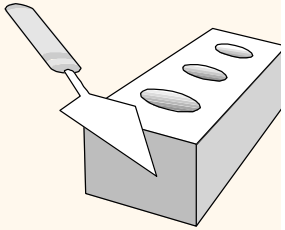
- Syntax :

```
TYPE  <nom_type>  IS  RECORD  (
        <nomchamps>  <type>  [ [NOT NULL]
                                [ :=  |  DEFAULT  <expression> ],
        . . . ) ;
```

- Example :

```
SQL>    DECLARE
        TYPE client_t IS RECORD (
                numero    NUMBER(4),
                nom                CHAR(20),
                adresse            CHAR(20) ) ;
        client1_v   client_t ;

        BEGIN
        client1_v.numero := 2516 ;
        END ;
        /
```

%TYPE

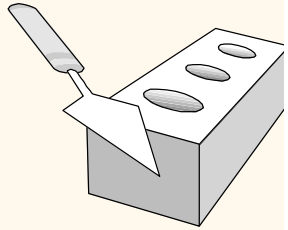


- Employed in variable declaration while re-using:
  - the definition of an attribute of an existing table
  - the definition of another variable declared previously
- Example :

```
. . .
nomEmploye_v          employe.nomEmp%TYPE ;
solde_v                NUMBER(7, 2) ;
soldeMinimal_v         solde_v%TYPE := -2000 ;
```

- Note :
  - %ROWTYPE : as %TYPE but to define a variable of the recording type whose fields correspond to all the attributes of a table
  - constraints NOT NULL of the definition of the attributes of tables are not re-used with %TYPE

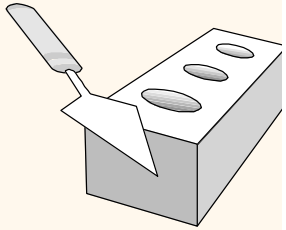
Display from a PL/SQL block: DBMS_OUTPUT package



- Set SERVEROUTPUT variable

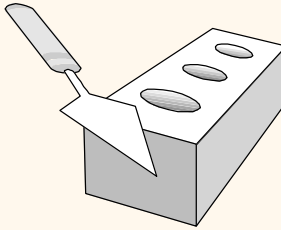
  SET SERVEROUTPUT ON
- Then use DBMS_OUTPUT package
- Main procedure of DBMS_OUTPUT package
  - put: add text on current line
  - new_line: carriage return
  - put_line: put + new_line
  - get_line: read a line
- Example

  DBMS_OUTPUT.PUT_Line('hello' ||user || '!');



Control Structures

- Conditional Control
  - IF - THEN - END IF
  - IF - THEN - ELSE - END IF
  - IF - THEN - ELSIF - END IF
- Loops
  - LOOP - END LOOP
  - WHILE - END LOOP
  - FOR - END LOOP
- Note : use order EXIT to leave any type of loops



Conditional Control

- Conditional Control :
- Syntax :

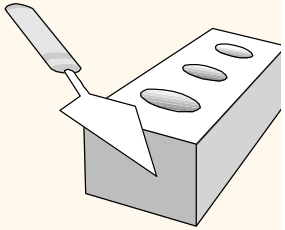
  IF condition THEN treatment;

  { ELSEIF condition THEN treatment; }

  [ELSE treatment;]

  END IF;

- Operators used : =, <, >, !=, <=, >=, IS NULL, IS NOT NULL, BETWEEN, LIKE, AND, OR, …

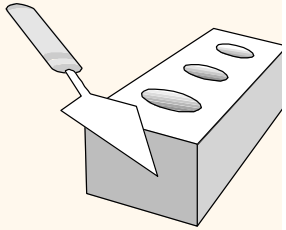


Multiple Conditional (CASE)

```
CASE expression
WHEN value THEN commands
WHEN value THEN commands;
…
[ELSE commands;]
END CASE;
```

- Exemple:

```
CASE note
WHEN ‘A’ THEN dbms_output.put_line (‘good’);
WHEN ‘B’ THEN dbms_output.put_line (‘average’);
WHEN ‘C’ THEN dbms_output.put_line (‘bad’);
ELSE dbms_output.put_line (‘mark not found’);
END CASE;
```

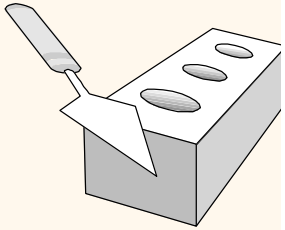


Iterative Control

- Iterative Control : LOOP
- Syntax :

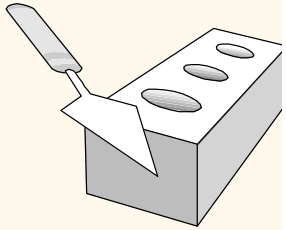
```
LOOP [ << label_name>>]
        [orders;]
   [ EXIT [label_name] WHEN condition ]
   [orders;]
END LOOP [label_name];
```

- Note : Without order EXIT, the loops LOOP are infinite



Loop - Example

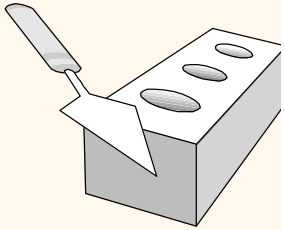
```
DECLARE
        fact NUMBER := 1;
        i NUMBER := 1;
BEGIN
LOOP
      fact := fact * i;
      i := i+1;
      EXIT  WHEN i = 10;
END LOOP;
INSERT INTO resultat
                VALUES ('fact(9) = ', fact);
END ;
/
```



Iterative Control

- Iterative Control : WHILE
- Syntax :

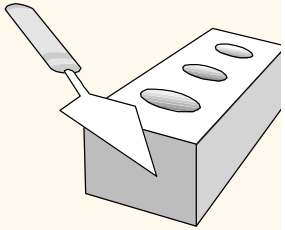
```
[<< label_name >>]
WHILE condition
LOOP
   orders;
END LOOP [label_name];
```



While - Example

```
DECLARE
        fact NUMBER := 1;
        i    NUMBER := 1;
BEGIN
WHILE i <=9
LOOP
        fact := fact * i;
        i := i+1;
END LOOP;
INSERT INTO resultat
                VALUES ('fact(9) = ', fact);
END ;
/
```

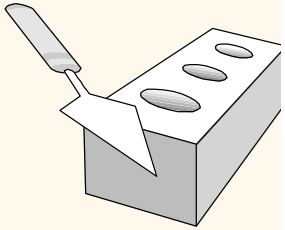
Iterative Control



- Iterative Control : FOR
- Syntax :

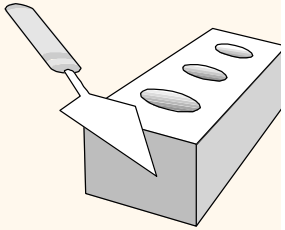
```
[<< label_name >>]
FOR identifier IN [REVERSE] exp1 ..exp2
LOOP
      orders;
END LOOP [label_name];
```

- The identifier is declared implicitly



For - Example

```
DECLARE
      fact NUMBER := 1;
BEGIN
FOR i IN 1..9
LOOP
      fact := fact * i;
END LOOP;
INSERT INTO resultat
      VALUES ('fact(9) = ', fact);
END ;
/
```

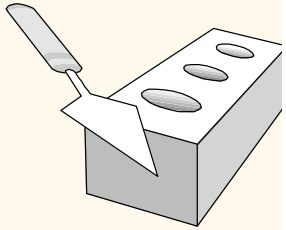


Control Structures

- Do not modify the identifier of a loop FOR
- The loops can be overlapping
- One can name the loops to identify explicitly which of the two overlapping loops finish

```
. . .
<<ExternalLoop>>
LOOP
  . . .
  EXIT WHEN compteur_v = 10 ;
  <<InternalLoop>>
  LOOP
            EXIT ExternalLoop WHEN compteur_v = 100 ;
             EXIT InternalLoop WHEN drapeau_v = TRUE ;
  END LOOP InternalLoop ;
END LOOP ExternalLoop;
. . .
```

SQL Orders in PL/SQL



- SELECT :

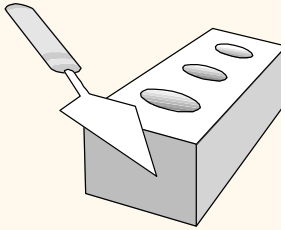
  SELECT attribute, …

  INTO list of variables

  FROM table

  [WHERE condition]

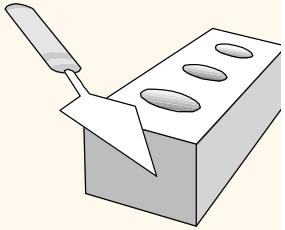
- The SQL query must return only one record
- If it is not the case, exceptions NO_DATA_FOUND or TOO_MANY_ROWS are raised.



Update data with PL/SQL

- Three orders of the data manipulation language (DML) of SQL make it possible to modify a data base :
  - INSERT
  - UPDATE
  - DELETE

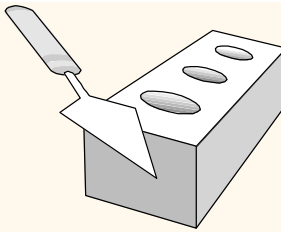
Update data with PL/SQL (UPDATE)



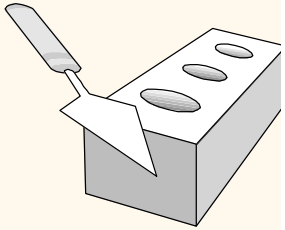
```
DECLARE
   majorationSalaire_v        Employe.salaire%TYPE := 2000;
BEGIN
   UPDATE          Employe
   SET       salaire = salaire + majorationSalaire_v
   WHERE job = 'PROGRAMMEUR' ;
END ;
```

- Note :
  - if a variable has the same name as a name of an attribute of the table handled in clause WHERE, the Oracle server uses in priority the attribute of the table

Update data with PL/SQL (DELETE)



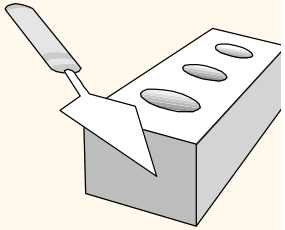
```
DECLARE
   noDept_v Employe.noDept%TYPE := 10 ;
BEGIN
   DELETE FROM Employe
            WHERE noDept = noDept_v ;
END ;
/
```



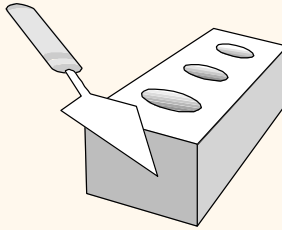
Transactions with PL/SQL

- First order INSERT/UPDATE/DELETE of a block starts a new transaction
- The end of the block finishes the transaction
- To finish a transaction explicitly, it's necessary to use orders SQL:
  - COMMIT : validate the modifications made since the beginning of the transaction in progress, and starts a new transaction
  - ROLLBACK : cancel all the modifications made since the beginning of the transaction in progress, and starts a new transaction

Transactions with PL/SQL



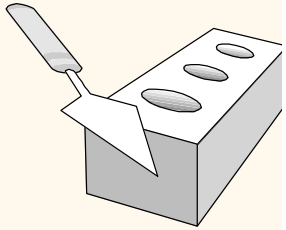
```
DECLARE
   noDept_v Employe.noDept%TYPE := 10 ;
   majorationSalaire_v Employe.salaire%TYPE := 2000;
BEGIN
   DELETE FROM Employe
               WHERE noDept = noDept_v ;
   COMMIT ;
   UPDATE Employe
   SET       salaire = salaire + majorationSalaire_v
   WHERE job = 'PROGRAMMEUR' ;
END ;
/
```



Procedures

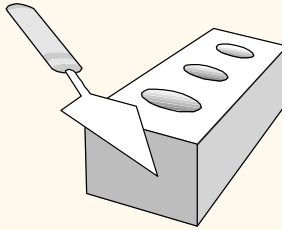
- Creation of a procedure:
  CREATE [OR REPLACE] PROCEDURE procedure_name
  [argument [mode] type,…]
  [IS | AS] block PL/SQL

- There are three types of parameters that can be declared:
  - **IN** - The value of the parameter can not be overwritten by the procedure.
  - **OUT** – The value of the parameter can be overwritten by the procedure.
  - **IN OUT** - The parameter can be referenced by the procedure and the value of the parameter can be overwritten by the procedure.



Procedure Example

```
CREATE OR REPLACE PROCEDURE conversion_FF_euro
(price_FF IN REAL, price_euro OUT REAL)
IS
        rate CONSTANT REAL := 6.55957;
BEGIN
        IF price_FF IS NOT NULL THEN
                price_euro := price_FF/rate;
        ELSE
                dbms_output.put_line ('conversion not possible');
        END IF;
END conversion_FF_euro ;
```



Function

- Creation of a function :

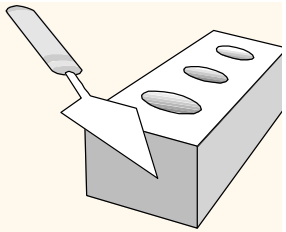
  CREATE [OR REPLACE] FUNCTION nom_fonction

  [argument [IN] type, …]]

  RETURN return_type

  [IS | AS] block PL/SQL

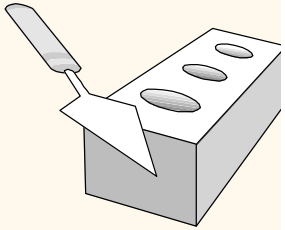
- where
  - **RETURN**
    Introduces the RETURN clause, which specifies the datatype of the return value.



(recursive) Function Example

```
CREATE FUNCTION factorial (n INTEGER)
RETURN INTEGER
IS
BEGIN
        IF n=1 THEN
                RETURN 1;
        ELSE
                RETURN n*factorial (n-1);
        END IF;
END;
```

Notes about procedures and functions

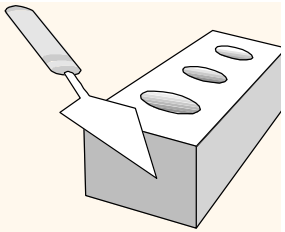


- To find out what procedures and functions you have created, use the following SQL query:

```
select object_type, object_name from
  user_objects where object_type =
  'PROCEDURE'    or object_type =
  'FUNCTION';
```

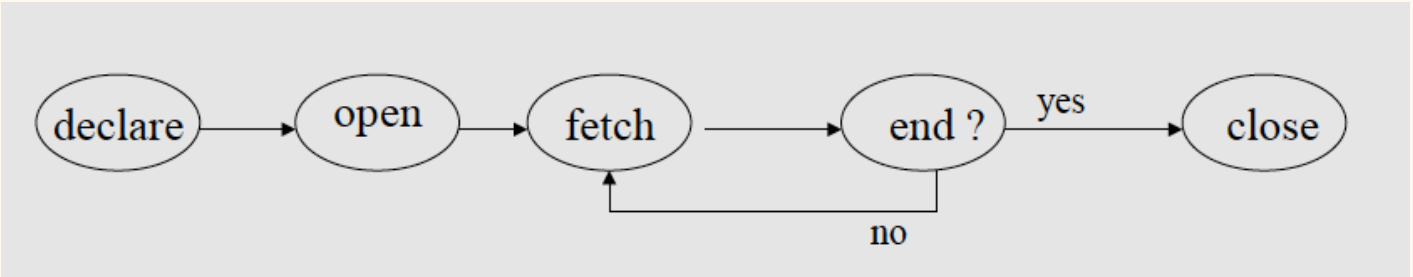
- To drop a stored procedure/function:

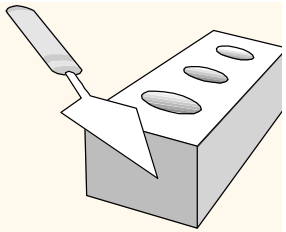
```
drop procedure <procedure_name>;
drop function <function_name>;
```



Cursor

- A cursor is a kind of pointer which enables browsing the result of a query, tuple by tuple :
  - Declaration of the cursor (CURSOR IS)
    - A SELECT query is associated to the cursor
    - No visible effect
  - Cursor Opening (OPEN)
    - The SELECT query is evaluated
    - The cursor points to the first tuple
  - Reads the current tuple and moves to next tuple (FETCH)
  - Cursor close (CLOSE)
- Two types of cursors:
  - implicit cursors
    Oracle server use implicit cursors to execute SQL queries
  - Explicit Cursors
    variables explicitly declared by the programmer



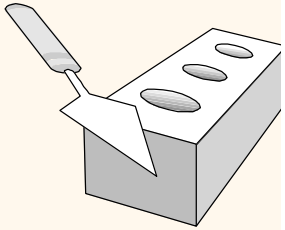


Cursor Example

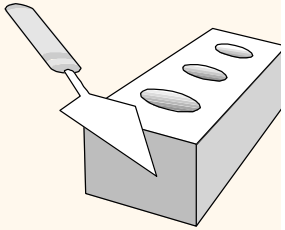
```
1)      DECLARE
 /* Output variables to hold the result of the query: */
2)    a T1.e%TYPE;
3)    b T1.f%TYPE;        /* Cursor declaration: */
4)    CURSOR T1Cursor IS
5)       SELECT e, f
6)       FROM T1
7)       WHERE e < f
8)       FOR UPDATE;
9) BEGIN
10)    OPEN T1Cursor;
11)    LOOP          /* Retrieve each row of the result of the above query          into PL/
       SQL variables: */
12)       FETCH T1Cursor INTO a, b;          /* If there are no more rows to fetch, exit the
       loop: */
13)       EXIT WHEN T1Cursor%NOTFOUND;          /* Delete the current tuple: */
14)       DELETE FROM T1 WHERE CURRENT OF T1Cursor;          /* Insert the reverse
       tuple: */
15)       INSERT INTO T1 VALUES(b, a);
16)    END LOOP;       /* Free cursor used by the query. */
17)    CLOSE T1Cursor;
18) END; /
```

```
CREATE TABLE T1(
        e INTEGER,
        f INTEGER );
```

Cursors



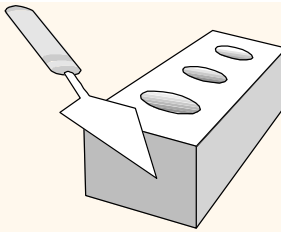
- Attributes of the cursors: by using the attributes of cursors, you can test the result of the SQL query
  - SQL%ROWCOUNT
    - number of tuples already processed
  - SQL%FOUND
    - Boolean, TRUE if the last SQL query has affected more than one tuple
  - SQL%NOTFOUND
    - Boolean, TRUE if the last SQL query does not have affected any tuple
  - SQL%ISOPEN
    - Boolean indicating if the cursor is opened or closed (by default, implicit cursors are always closed at the end of the query)
- Note : in the place of ' SQL ', use the name of your cursor to identify explicit cursor



Explicit cursors

- Declaration : in the section DECLARE
  - CURSOR name_cursor IS order_select;
- Open : in the section BEGIN .. END
  - OPEN name_cursor;
- Assignment of the values of a row to the receiving variables or the structure (often in a loop)
  - FETCH name_cursor INTO variables /record;
- Closing and release of the memory:
  - CLOSE name_cursor;

Cursors



```
DECLARE
    CURSOR departementVentes_v IS
        SELECT      *
        FROM        Departement
        WHERE       nomDept = 'VENTES' ;
    unDepartement_v     Departement%ROWTYPE ;
    compteur_v  number := 0 ;
BEGIN
    OPEN departementVentes_v  ;
    LOOP
        FETCH  departementVentes_v  INTO unDepartement_v ;
        EXIT WHEN departementVentes_v%NOTFOUND ;
        compteur_v := compteur_v +1 ;
    END LOOP ;
    CLOSE departementVentes_v ;
END ;
 /
```

déclaration

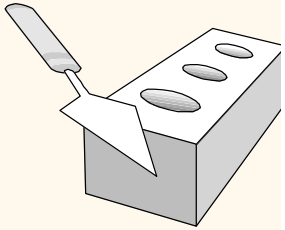
ouverture

utilisation

utilisation

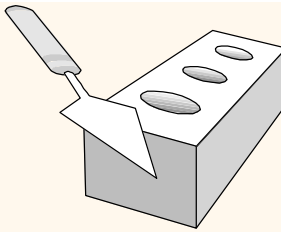
fermeture

Trigger

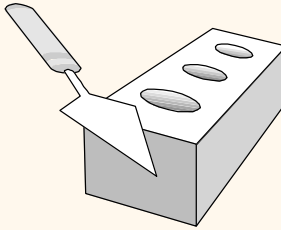


- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Enables defining dynamic constraints
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)

Trigger

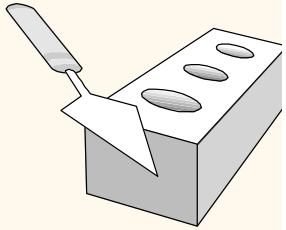


CREATE [OR REPLACE] TRIGGER
<trigger_name>
{BEFORE | AFTER} {INSERT | DELETE |
UPDATE} ON <table_name>
[REFERENCING [NEW AS <new_row_name>]
[OLD AS <old_row_name>]]
[FOR EACH ROW [WHEN
(<trigger_condition>)]]
<trigger_body (PL/SQL)>



Trigger : Example

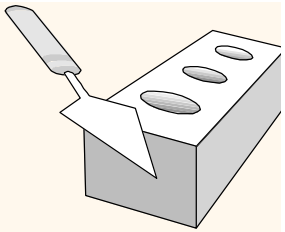
```
CREATE OR REPLACE TRIGGER maintienDuStock
AFTER INSERT OR UPDATE ON Stock
FOR EACH ROW
WHEN (:new.quantiteDispo < 10 OR :new.quantiteDispo IS NULL)
    BEGIN
        INSERT INTO Commandes (N°Produit, quantiteCommandee)
            VALUES (:new.N°Produit, 200) ;
    END ;
    /
```



Triggers: some important points

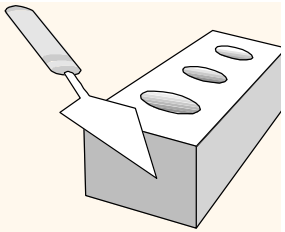
- Only for row-level triggers:
  - The special variables NEW and OLD are available to refer to new and old tuples respectively. **Note:** In the trigger body, NEW and OLD must be preceded by a colon (":"), but in the WHEN clause, they do not have a preceding colon!
  - The REFERENCING clause can be used to assign aliases to the variables NEW and OLD.

Trigger



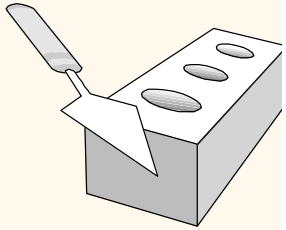
- To access the values of the attributes of the modified line
  - Use of two variables :
    - **:old**
    - **:new**
- **For an « INSERT » Trigger**
  - the new values are in :new.<attribute_name>
- **For an «UPDATE» Trigger**
  - the old values are in :old.<attribute_name>
  - the new values are in :new.<attribute_name>
- **For a «DELETE» Trigger DELETE**
  - the old values are in :old.<attribute_name>

Triggers BEFORE and AFTER

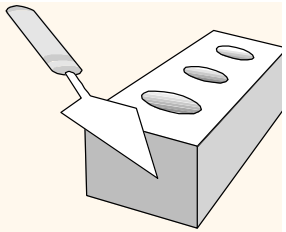


- Trigger BEFORE row level:
  - is executed before the triggering event takes place
  - may affect the values of the inserted or modified row
- Trigger AFTER row level:
  - is executed after the triggering event takes place
  - can not affect the values of the inserted or modified row
- Six possible triggers:
  - BEFORE INSERT, BEFORE UPATE, BEFORE DELETE
  - AFTER INSERT, AFTER UPDATE, AFTER DELETE

Activation / disabling / delete triggers



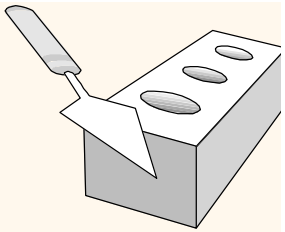
- Disabling a trigger :
  - ALTER TRIGGER <trigger_name> DISABLE ;
- Activation of the trigger :
  - ALTER TRIGGER <trigger_name> ENABLE ;
- Delete trigger :
  - DROP TRIGGER <trigger_name> ;



Notes about triggers

- **Cascading Triggers:**
  Trigger performing INSERT, UPDATE or DELETE can generate events leading to the execution of one or more other triggers. This is known as cascading triggers. Avoid more than two-level cascading triggers.
- **Mutation:**
  When a modification (INSERT, UPDATE or DELETE) in a table is not validated by a COMMIT, it is called « mutatig table ». A row-level trigger can not read or modify a mutating table.
- **Validation:**
  A trigger can not execute COMMIT or ROLLBACK, or call a function, procedure or a package

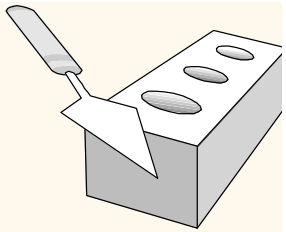
CYCLIC CASCADING in a TRIGGER



- This is an undesirable situation where more than one trigger enter into an infinite loop. while creating a trigger we should ensure the such a situtation does not exist.

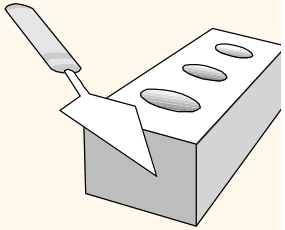
- The below example shows how Trigger's can enter into cyclic cascading. Let's consider we have two tables 'abc' and 'xyz'. Two triggers are created.
  1) The INSERT Trigger, triggerA on table 'abc' issues an UPDATE on table 'xyz'.
  2) The UPDATE Trigger, triggerB on table 'xyz' issues an INSERT on table 'abc'.

- In such a situation, when there is a row inserted in table 'abc', triggerA fires and will update table 'xyz'.
  When the table 'xyz' is updated, triggerB fires and will insert a row in table 'abc'.
  This cyclic situation continues and will enter into a infinite loop, which will crash the database.



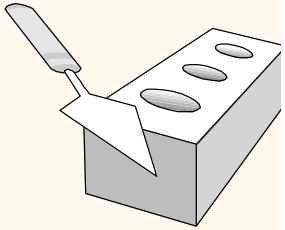
Exceptions

- The management of the exceptions makes it possible to affect a treatment appropriate to the errors which have occurred during the execution of a block PL/SQL.
- 2 types :
  - Internal error Oracle: exceptions thrown by Oracle all have a predefined number and an associated message
  - Error in the program of the user
- The exceptions are treated in a particular section
  - Allows not to have to check the errors at every moment
  - Separate the normal treatment from the treatment associated with the situations with error
- When an exception is thrown:
  - The PL/SQL block is automatically terminated
  - Instructions associated with exception processing block are executed



Exceptions

- Exceptions can be internally defined (by the runtime system) or user defined
- Examples of internally defined exceptions include *division by zero* and *out of memory*. Some common internal exceptions have predefined names, such as ZERO_DIVIDE.
- You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named insufficient_funds to flag overdrawn bank accounts.

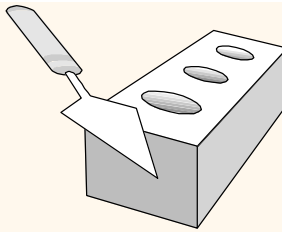


To raise explicitely an Exception

- Syntax :
  <exception_name> EXCEPTION ;
- To associate a number to an exception (in the declarations section)

  PRAGMA EXCEPTION_INIT (< exception_name >,< number>) ;
- To raise the exception explicitly:
  RAISE < exception_name > ;
- Note :
  - the numbers from 0 to -20000 are reserved to implicit exceptions
  - use the numbers from -20000 to -20999

Exceptions

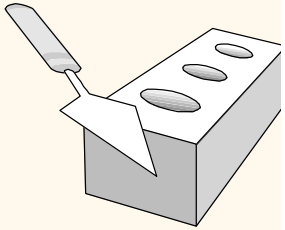


DECLARE
o o o
BEGIN
o o o
EXCEPTION
o o o
END ;

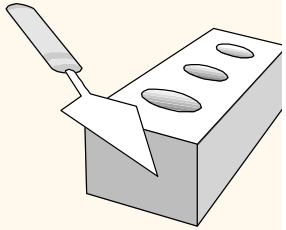
- Syntax :

```
EXCEPTION
    WHEN <exception 1> [OR <exception
    2> . . . ] THEN
      <instructions>
      . . .
    WHEN <exception 3> [OR <exception
    4> . . . ] THEN
      <instructions>
      . . .
    WHEN OTHERS THEN
      <instructions>
    END ;
```

Define, raise and handle an exception: Example



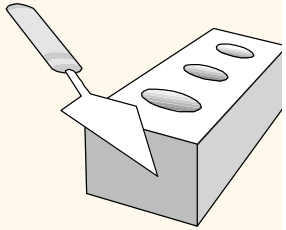
```
DECLARE
    erreurDonnees_v EXCEPTION ;
    PRAGMA EXCEPTION_INIT (erreurDonnees_v, -2292) ;
BEGIN
    . . .
    IF (noDept_v > 10) THEN
      RAISE erreurDonnees_v  ;
    END IF;
. . .
EXCEPTION
    WHEN erreurDonnees_v THEN
       . . .
END ;
/
```



Exceptions

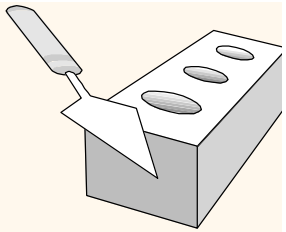
- To recover the numeric code of the exception which has been raised:
  - SQLCODE
- To retrieve the corresponding message
  - SQLERRM
- Example :

```
BEGIN
        dbms_output.enable ;
        . . .
EXCEPTION
        WHEN OTHERS THEN
                dbms_output.put_line (‘code ’ || TO_CHAR(SQLCODE)) ;
                dbms_output.put_line (SQLERRM) ;
END ;
```



Exceptions

- Examples :
  - ZERO_DIVIDE
  - INVALID_CURSOR
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - CURSOR_ALREADY_OPEN
  - VALUE_ERROR
  - LOGIN_DENIED
  - INVALID_NUMBER
  - VALUE_ERROR
  - ZERO_DIVIDE
  - …



Defining Your Own Error Messages: Procedure RAISE_APPLICATION_ERROR

- Syntax :

  RAISE_APPLICATION_ERROR (<number>, <message>)

- Example :

```
DECLARE
  erreurDonnees_v EXCEPTION ;
BEGIN
    . . .
    EXCEPTION
          WHEN erreurDonnees_v THEN
          RAISE_APPLICATION_ERROR (-20000, 'Données non
    valides');
END ;
```