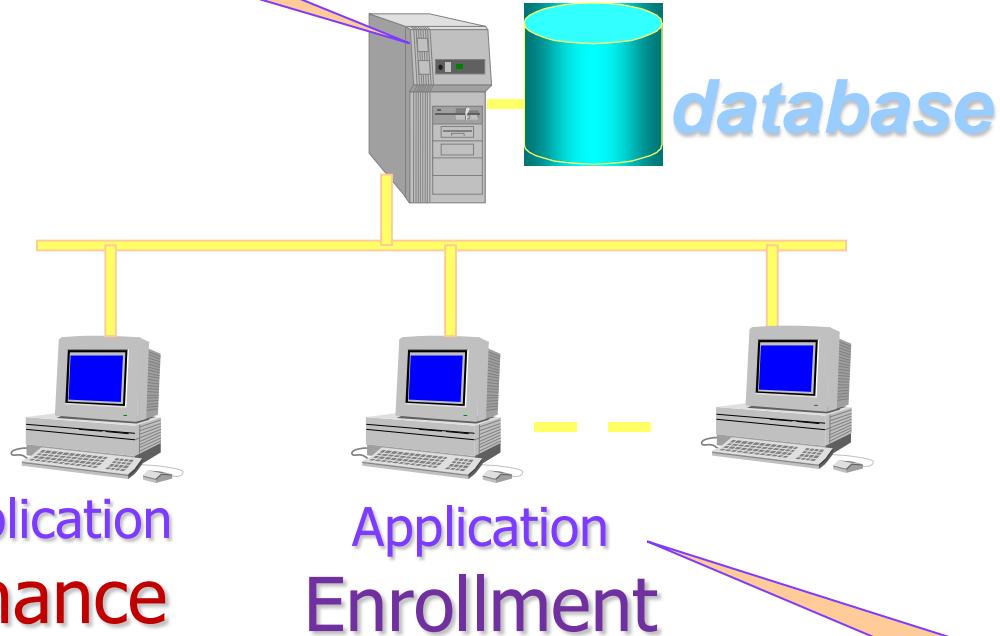


Stored Procedures & Triggers



DBMS

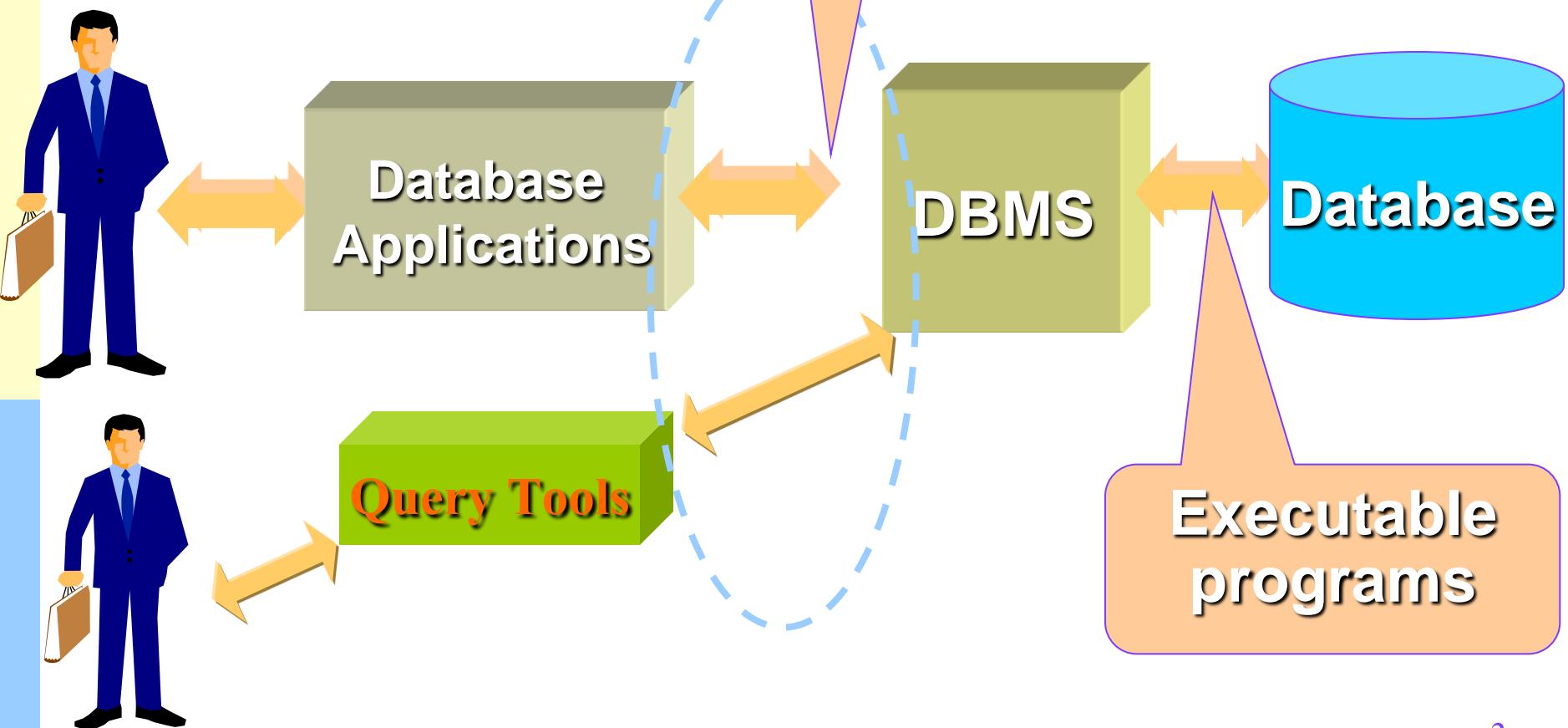


- ***Multiple programs***
- ***Many types of users***

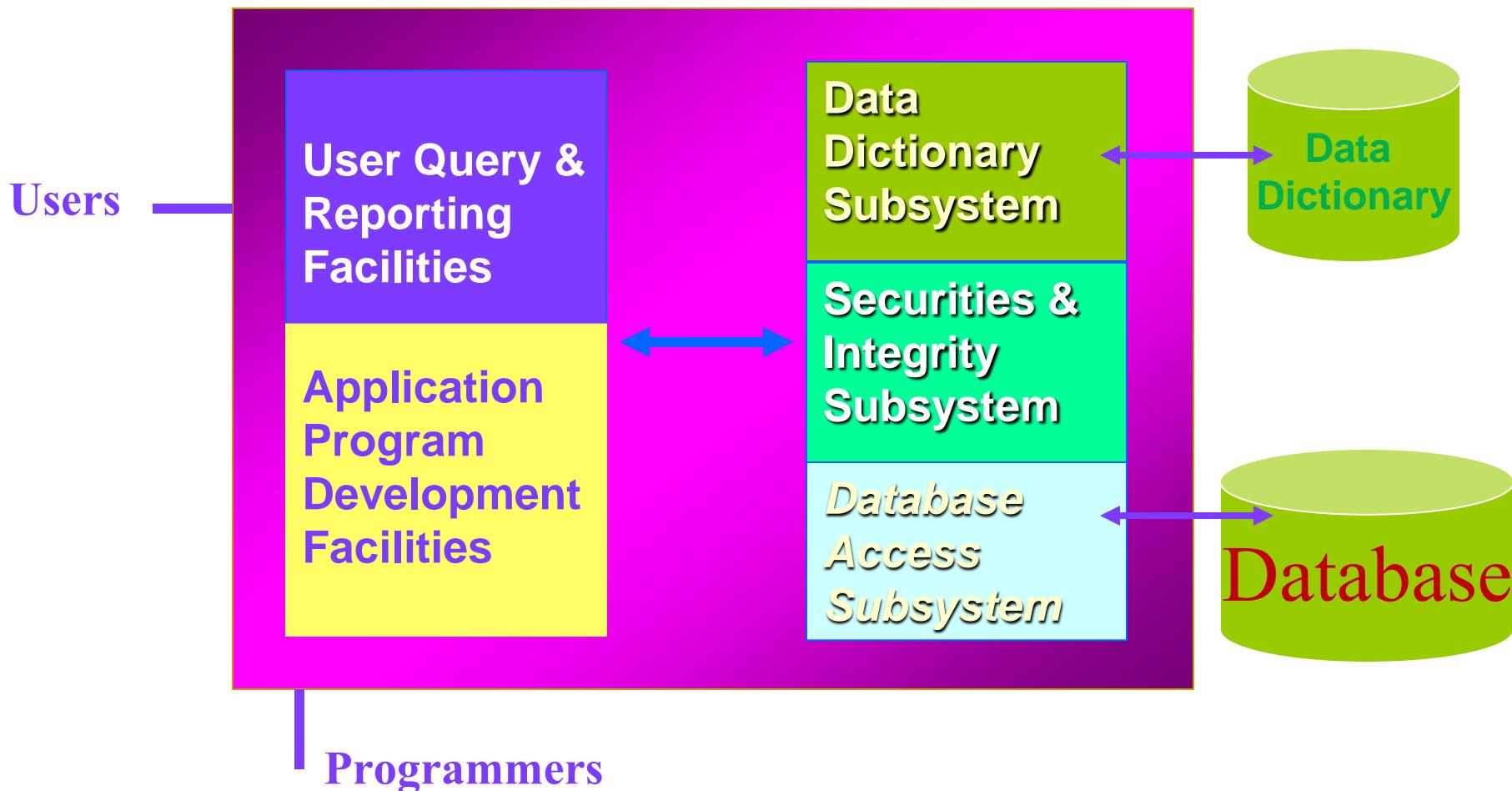
Application development tools

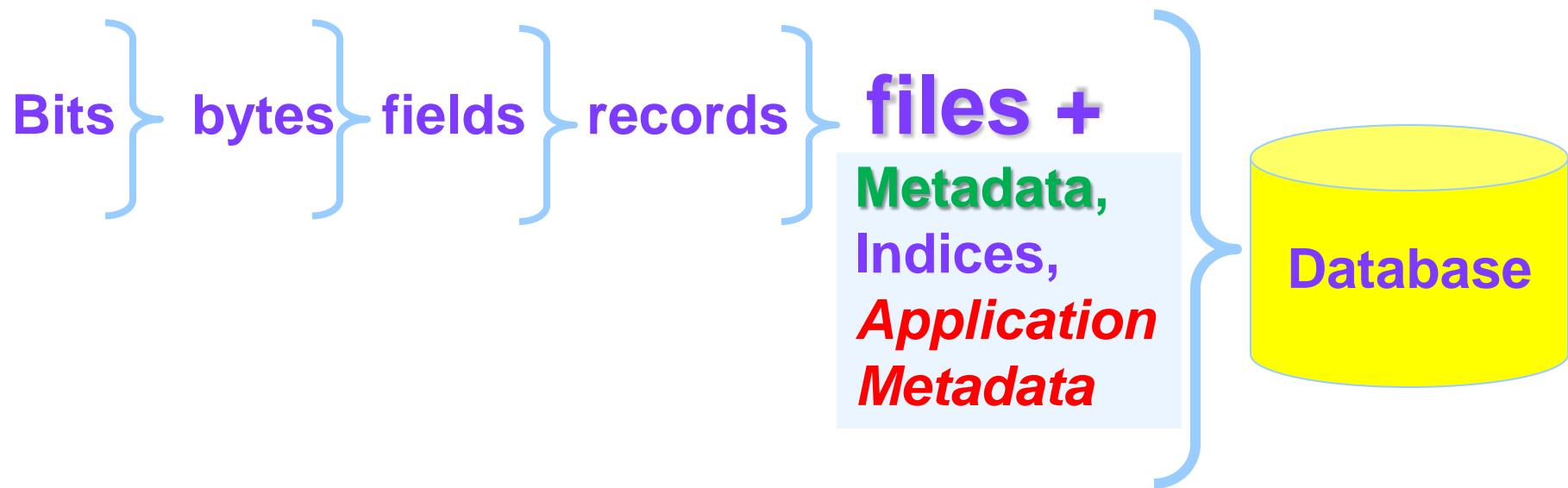
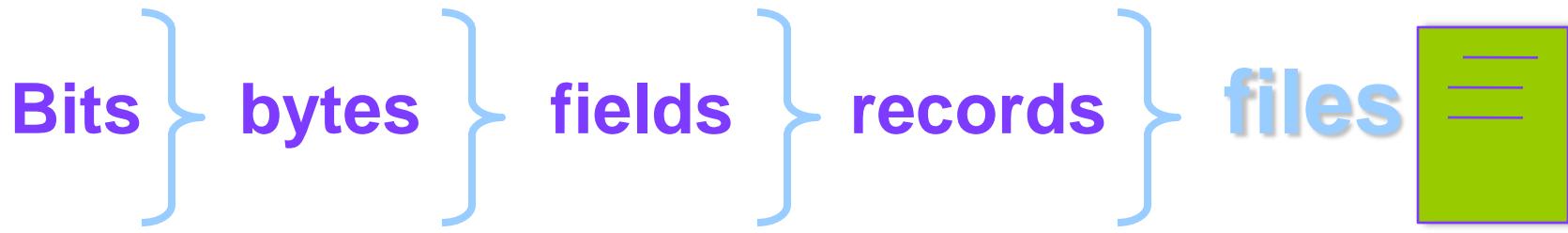
Database access

SQL
commands



DBMS





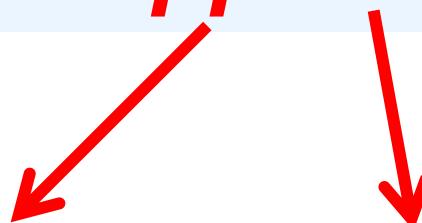
Database

files +

- 1. Metadata,
- 2. Indices,
- 3. Application Metadata**

Application
Finance

Application
Enrollment



Outline

- *Database programming language background*
- **Stored *procedures***
- **Triggers**

Motivation for Database Programming Languages

- Customization
- Batch processing
- Complex operations
- Efficiency and portability

Database Programming Languages

- A procedural language with an interface to one or more DBMSs.
- Interface allows a program to combine
 - *procedural statements* with
 - *nonprocedural database access (SQL)*.

Design Issues

- Language style: call-level vs. statement-level interface
- Binding: static vs. dynamic
- Database connection: implicit vs. explicit
- Result processing: data types and processing orientation

Language Style

- Call-level interface:
 - a set of procedures and
 - a set of type definitions

for manipulating the results of SQL statements
- ODBC and JDBC are widely used call-level interfaces

Language Style

- ❑ Statement-level interface: changes to the syntax of a host programming language to accommodate embedded SQL statements
- ❑ Most DBMSs support both interfaces

Binding

- Association of access plan with an SQL statement
- Static binding: association at compile time
- Dynamic binding: association at run time

Binding options

- Statement-level interface
 - Static
 - dynamic
- Call-level interface
 - Dynamic
- Reuse of access plans for repetitively executed statements in a program

Database Connection

- Implicit for stored procedures and triggers because they are part of a database
- External programs: explicit connection
 - CONNECT statement or procedure
 - Web address or database identifier
 - Database identifier is more flexible

SQL Results Processing

- The data types in a programming language may not correspond exactly to the standard SQL data types
- Need data type mapping
 - mapping statements or procedures

JDBC type	Java type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

SQL Results Processing

- ❑ Processing orientation
 - SELECT statements for single row results
 - ❑ result values can be stored in program variables

Select INTO variables

DECLARE

- ❑ deptid employees.department_id%TYPE;
- ❑ jobid employees.job_id%TYPE;

SELECT department_id, job_id

- ❑ **INTO** deptid, jobid
- ❑ **FROM** employees
- ❑ **WHERE** employee_id = 140;

SQL Results Processing

- ❑ Processing orientation
 - Cursor for multiple row results
 - Cursor is similar to a dynamic array
 - ❑ array size = size of the query result
- ❑ Interface provides statements to declare, open, close, iterate (position), and retrieve values

Overview of PL/SQL

- Proprietary database programming language for Oracle
- Widely used language
- Java style syntax with a statement level interface
- Use PL/SQL for writing stored procedures and triggers

User Identifiers in PL/SQL

- Provide **names** for variables and constants
- Not case sensitive
- Restrictions
 - At most 30 characters
 - Must begin with a letter
 - Must be unique
 - Allowable characters are letters, numbers, _, #, and \$

PL/SQL Constants

- Numeric constants: whole numbers, fixed decimal numbers, and scientific notation
- String constants: use single quotes; case sensitive
- Boolean constants: TRUE, FALSE
- NULL: constant for every data type
- No string constants: use the To_Date function to create string constants

PL/SQL Data Types

- String: CHAR(L), VARCHAR2(L)
- Numeric: INTEGER, DECIMAL(W,D),
FLOAT(P). SMALLINT
- Logical: BOOLEAN
- DATE: stores both date and time

Variable Declaration Examples

DECLARE

```
aFixedLengthString      CHAR(6) DEFAULT 'ABCDEF';
aVariableLengthString   VARCHAR2(30);
anIntegerVariable       INTEGER := 0;
aFixedPrecisionVariable DECIMAL(10,2);
-- Uses the SysDate function for the default value
aDateVariable          DATE DEFAULT SysDate;
-- Anchored declarations
anOffTerm Offering.OffTerm%TYPE;
anOffYear Offering.OffYear%TYPE;
aCrsUnits Course.CrsUnits%TYPE;
aSalary1 DECIMAL(10,2);
aSalary2 aSalary1%TYPE;
```

Assignment Examples

```
aFixedLengthString := 'XYZABC';  
anIntegerVariable   := anAge + 1;  
  
aFixedPrecisionVar := aSalary*0.10;
```

String concatenation function (||)

```
aVarLengthString := aFixedLengthString ||  
    'ABCDEF';
```

-- To_Date is the date conversion function

```
aDateVariable := To_Date('30-Jun-2006');
```

IF Statement Format

IF-THEN Statement:

```
IF condition THEN  
    sequence of statements  
END IF;
```

IF-THEN-ELSE Statement:

```
IF condition THEN  
    sequence of statements 1  
ELSE  
    sequence of statements 2  
END IF;
```

CASE Statement Format

CASE Statement (Oracle 9i/10g only):

CASE selector

WHEN expression1 THEN *sequence of statements 1*
WHEN expression2 THEN *sequence of statements 2*
WHEN expressionN THEN *sequence of statements N*

[ELSE *sequence of statements N+1*]

END CASE;

Formats of Iteration Statements

FOR LOOP Statement:

```
FOR variable IN BeginExpr .. EndExpr LOOP  
    sequence of statements  
END LOOP;
```

WHILE LOOP Statement:

```
WHILE condition LOOP  
    sequence of statements  
END LOOP;
```

LOOP Statement:

```
LOOP  
    sequence of statements containing an EXIT statement  
END LOOP;
```

Common SQL *Plus Commands

- **CONNECT**: login to a database
- **DESCRIBE**: list table details
- **EXECUTE**: execute statements
- **HELP**: lists column details
- **SET**: assigns values to SQL *Plus environment variables
- **SHOW**: displays error details
- **SPOOL**: send output to a file

PL/SQL Blocks

- Anonymous blocks to test procedures and triggers
- Named blocks for stored procedures

Block Structure:

```
[ DECLARE  
  sequence of declarations ]  
BEGIN  
  sequence of statements  
[ EXCEPTION  
  sequence of statements to respond to exceptions ]  
END;
```

Anonymous Block Example

```
SET SERVEROUTPUT ON; -- SQL Plus command
```

```
-- Anonymous block
```

```
DECLARE
```

```
    TmpSum      INTEGER;  
    TmpProd     INTEGER;  
    Idx         INTEGER;
```

```
BEGIN
```

```
    TmpSum := 0;
```

```
    TmpProd := 1;
```

```
-- Use a loop to compute the sum and product
```

```
FOR Idx IN 1 .. 10 LOOP
```

```
    TmpSum := TmpSum + Idx;
```

```
    TmpProd := TmpProd * Idx;
```

```
END LOOP;
```

```
Dbms_Output.Put_Line('Sum is ' || To_Char(TmpSum));
```

```
Dbms_Output.Put_Line('Product is ' || To_Char(TmpProd));
```

```
END;
```

```
/
```

Motivation for Stored Procedures

- Compilation of programming language statements and SQL statements
- Management of dependencies by the DBMS
- Centralized management of procedures
- Development of more complex functions and procedures
- Usage of DBMS security system for stored procedures

Format of PL/SQL Procedures

```
CREATE [OR REPLACE] PROCEDURE ProcedureName
  [ (Parameter1, ..., ParameterN) ]
IS
  [ sequence of declarations ]
BEGIN
  sequence of statements
  [ EXCEPTION
    sequence of statements to respond to exceptions ]
END;
```

Simple Procedure Example

```
CREATE OR REPLACE PROCEDURE pr_InsertRegistration
(aRegNo IN Registration.RegNo%TYPE,
aStdSSN IN Registration.StdSSN%TYPE,
aRegStatus IN Registration.RegStatus%TYPE,
aRegDate IN Registration.RegDate%TYPE,
aRegTerm IN Registration.RegTerm%TYPE,
aRegYear IN Registration.RegYear%TYPE) IS
-- Create a new registration
BEGIN
    INSERT INTO Registration
        (RegNo, StdSSN, RegStatus, RegDate, RegTerm, RegYear)
    VALUES
        (aRegNo, aStdSSN, aRegStatus, aRegDate, aRegTerm,
         aRegYear);
    dbms_output.put_line('Added a row to the table');
END;
/
```

Registration.RegNo%TYPE



%TYPE indicates the same type
as

Registration.RegNo type

Exception Example

```
CREATE OR REPLACE PROCEDURE pr_InsertRegistration  
(aRegNo IN Registration.RegNo%TYPE,  
aStdSSN IN Registration.StdSSN%TYPE,  
aRegStatus IN Registration.RegStatus%TYPE,  
aRegDate IN Registration.RegDate%TYPE,  
aRegTerm IN Registration.RegTerm%TYPE,  
aRegYear IN Registration.RegYear%TYPE,  
aResult OUT BOOLEAN ) IS  
-- aResult is TRUE if successful, false otherwise.
```

BEGIN

```
    aResult := TRUE;  
    INSERT INTO Registration  
    (RegNo, StdSSN, RegStatus, RegDate, RegTerm, RegYear)  
    VALUES  
    (aRegNo, aStdSSN, aRegStatus, aRegDate, aRegTerm,  
     aRegYear);
```

EXCEPTION

```
    WHEN OTHERS THEN aResult := FALSE;
```

END;

OTHERS exception catches a variety of errors



❑ such as

- a violation of a primary key constraint
or
- a foreign key constraint

Common Predefined Exceptions

- ❑ Cursor_Already_Open
- ❑ Dup_Val_On_Index
- ❑ Invalid_Cursor
- ❑ No_Data_Found
- ❑ Rowtype_Mismatch
- ❑ Timeout_On_Resource
- ❑ Too_Many_Rows

Format of PL/SQL Functions

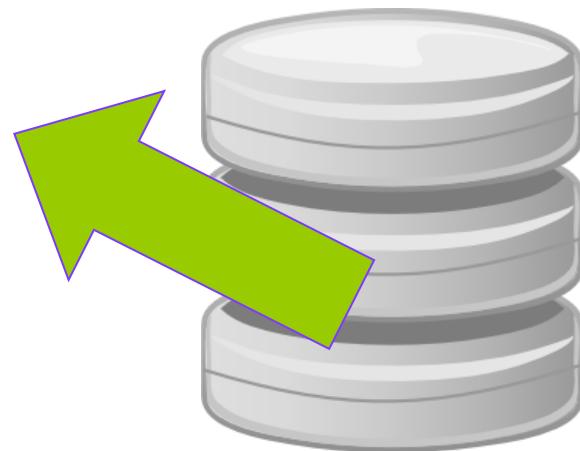
```
CREATE [OR REPLACE] FUNCTION FunctionName
[ (Parameter1, ..., ParameterN) ]
RETURN DataType
IS
[ sequence of declarations ]
BEGIN
sequence of statements including a RETURN statement
[ EXCEPTION
sequence of statements to respond to exceptions ]
END;
```

Simple Function Example

```
CREATE OR REPLACE FUNCTION fn_RetrieveStdName
(aStdSSN IN Student.StdSSN%type) RETURN VARCHAR2 IS
aFirstName Student.StdFirstName%type;
aLastName Student.StdLastName%type;
BEGIN
    SELECT StdFirstName, StdLastName
    INTO aFirstName, aLastName
    FROM Student
    WHERE StdSSN = aStdSSN;
    RETURN(aLastName || ', ' || aFirstName);
EXCEPTION
    WHEN No_Data_Found THEN
        RETURN(NULL);
    WHEN OTHERS THEN
        raise_application_error(-20001,'Database error');
END;
```

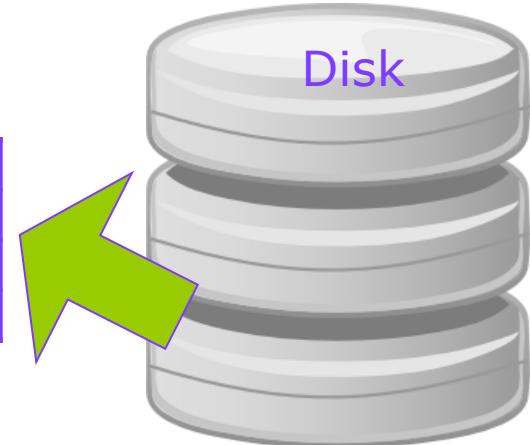
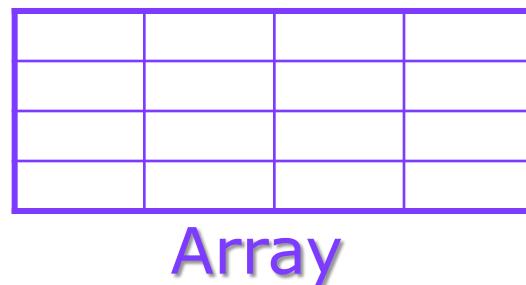


PL/SQL Cursors



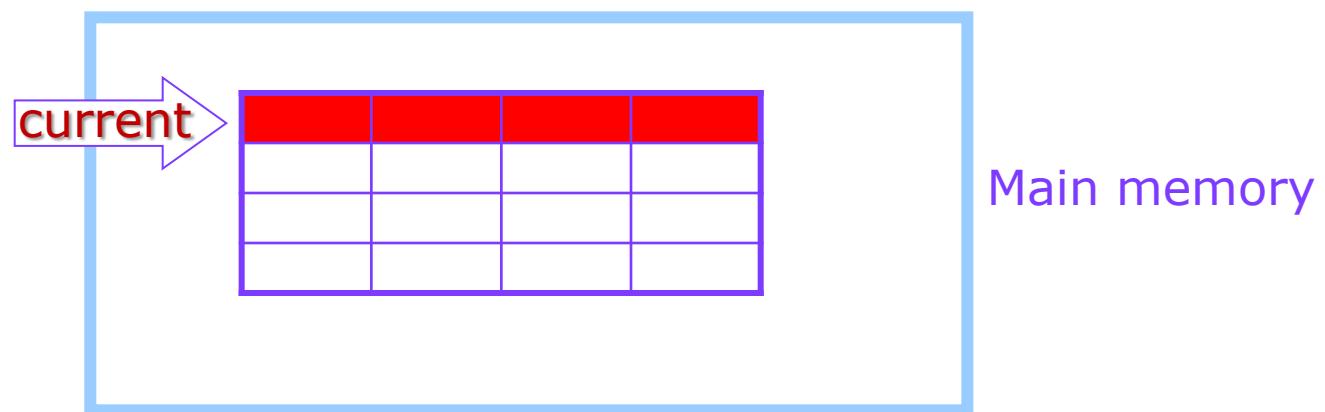
PL/SQL Cursors

- Supports usage of SQL statements that *return a collection of rows*
- Declaration statements
- *Specialized FOR* statement
- Cursor attributes (*indicate the status of cursor operations*)
- Actions on cursors



What are Cursors?

- A cursor is a temporary work area created in the system memory when a SQL statement is executed.
 - store the data retrieved from the database (*~array with a pointer*)



Classification of Cursors

- Implicit: declared, opened, and iterated inside a **special FOR statement**
 - Cursor created by DML :
 - **INSERT, UPDATE, DELETE, SELECT**
 - Return a single row
- Explicit: declared with the **CURSOR statement** in the **DECLARE section**
 - Cursor defined and manipulated by users

Classification of Cursors

- Statement binding:
 - Static: SQL statement specified at compile-time
 - Dynamic: SQL statement specified at execution

Common Cursor Attributes

Cursor states:

- ❑ %ISOpen:
 - true if cursor is open
- ❑ %Found:
 - true if cursor is not empty following a **FETCH statement**
- ❑ %NotFound:
 - true if cursor is empty following a **FETCH statement**
- ❑ %RowCount:
 - number of rows fetched

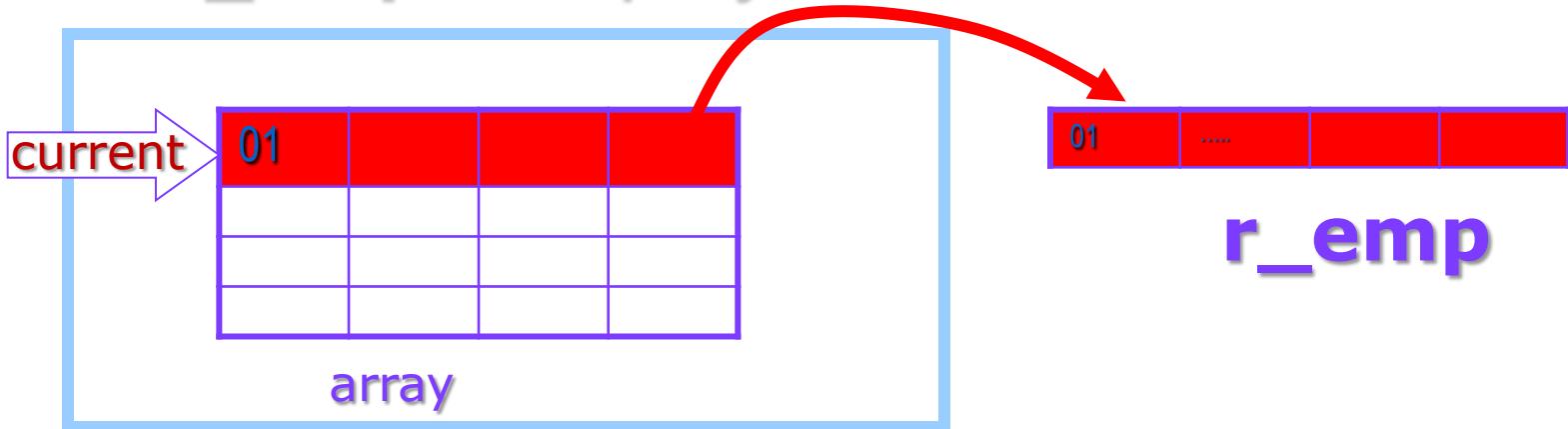
Returning an implicit cursor into a record *

```
declare
    r_emp    employee%ROWTYPE;
begin
    select * into r_emp from employee
    where id = '01';
    DBMS_OUTPUT.put_line(r_emp.first_Name);
end;
```

*if use select statement, avoid implicit cursor

□ declare

```
r_emp employee%ROWTYPE;
```



```
select * into r_emp from employee  
where id = '01';
```

Creating an explicit cursor

□ Syntax

```
CURSOR cursor_name IS  
    select_statement;
```

- *cursor_name* – A suitable name for the cursor.
- *select_statement* – A select query which returns multiple rows.

How to use Explicit Cursor?

There are 4steps in using an Explicit Cursor.

1. **DECLARE** the cursor in the declaration section.
2. **OPEN** the cursor in the Execution Section.
3. **FETCH** the data from cursor into PL/SQL variables or records in the Execution Section.
4. **CLOSE** the cursor in the Execution Section before you end the PL/SQL Block.

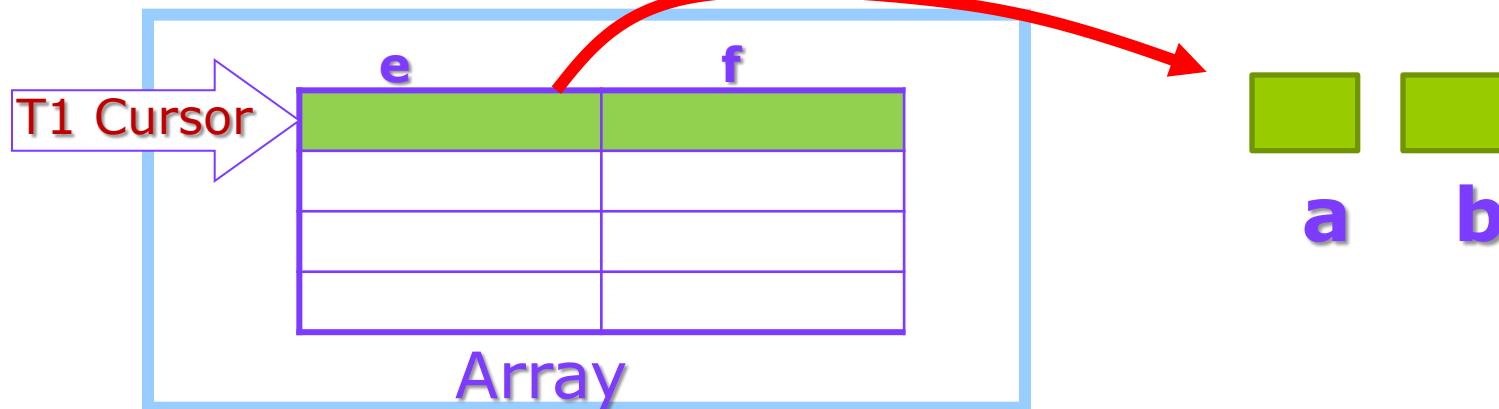
```
1. DECLARE      /* Output variables to hold the result of the query: */
2.   a T1.e%TYPE;
3.   b T1.f%TYPE;
4. CURSOR T1Cursor IS          /* Static Explicit Cursor declaration: */
5.   SELECT e, f
6.   FROM T1
7.   WHERE e < f
8.   FOR UPDATE;
9. BEGIN
10.  OPEN T1Cursor;
11.  /*Retrieve each row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

*/

DECLARE

a T1.e%TYPE;
b T1.f%TYPE;

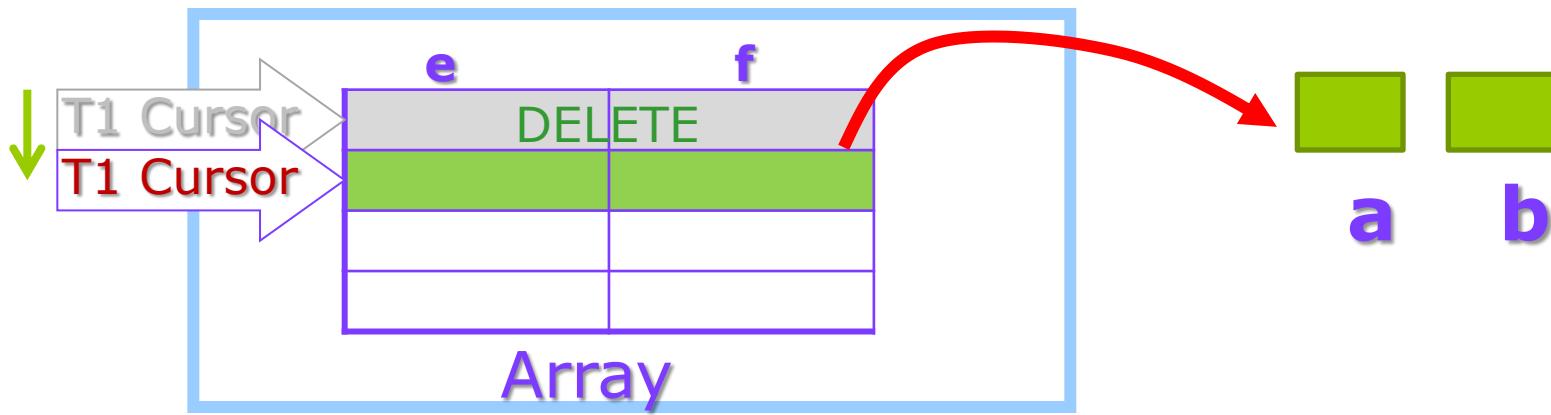
FETCH T1Cursor INTO a, b;



CURSOR T1Cursor IS
SELECT e, f
FROM T1
WHERE e < f
FOR UPDATE;

Next FETCH

FETCH T1Cursor INTO a, b;



LOOP

→ **FETCH T1Cursor INTO a, b;**

EXIT WHEN *T1Cursor%NOTFOUND*;

DELETE FROM T1 WHERE **CURRENT OF** T1Cursor;

INSERT INTO T1 VALUES(b, a);

END LOOP;

PL/SQL Packages

- Larger unit of modularity
- Improved reusability
- Groups *procedures, functions, exceptions, variables, constants, types, and cursors.*
- Public interface
- Private body: implementation of package
- Oracle provides predefined packages

Format of Package Interface

```
CREATE [OR REPLACE]
PACKAGE PackageName IS
[ Constant, variable, and type declarations ]
[ Cursor declarations ]
[ Exception declarations ]
[ Procedure definitions ]
[ Function definitions ]
END PackageName;
```

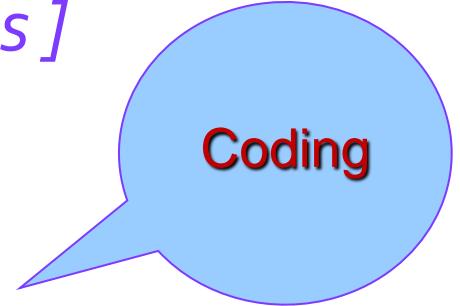


Declaration
only

All objects in a package interface are public.

Format of Package Body

```
CREATE [OR REPLACE]
PACKAGE BODY PackageName IS
    [Variable and type declarations]
    [Cursor declarations]
    [Exception declarations]
    [Procedure implementations]
    [Function implementations]
    [BEGIN sequence of statements]
    [EXCEPTION exception handling statements]
END PackageName;
```



Coding

Trigger



Trigger Overview

Triggers are rules managed by a DBMS.

- Triggers are executed by the rule system of the DBMS
- not by explicit calls as for procedures and functions.
- Ex.
 - execute procedure P1
 - Rule: after inserting a record in the student table

Trigger Overview

- Event-Condition-Action (ECA) rules
- Writing the action part or trigger body is similar to writing a procedure or a function
 - except that a trigger has no parameters.

Typical Usage of Triggers

- Complex integrity constraints
- Transition constraints
 - *compare the values before and after an update occurs*
- Update propagation (*in related tables*)
- Exception reporting
- Audit trail
 - *create a historical record of a transaction such as a history of automated teller usage*

Classification of Triggers

- **Granularity**

- Row: fire for each modified row
- Statement: fire once per SQL statement

- **Timing: before or after**

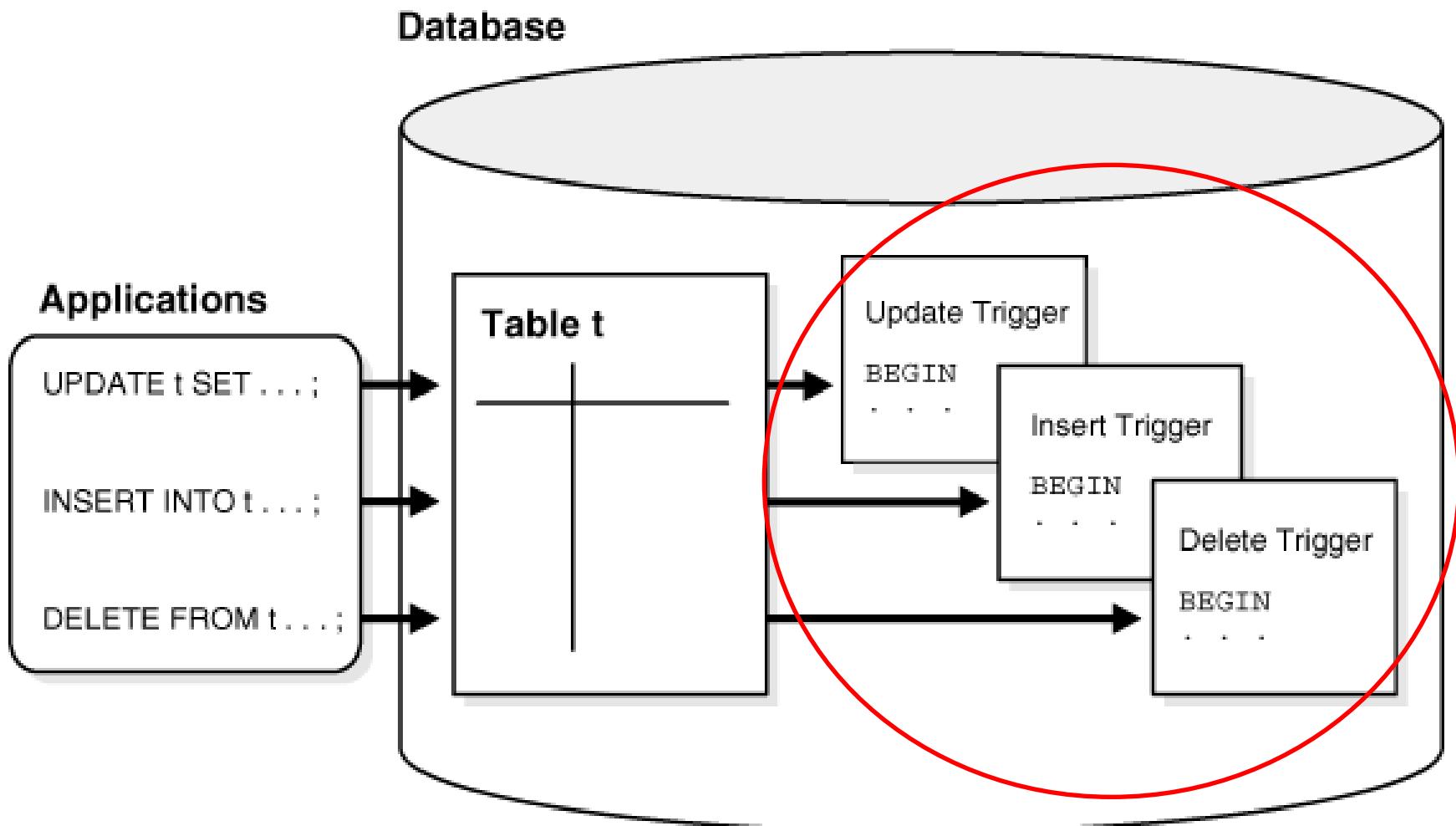
- **Event**

- Manipulation statements
- Update event with a list of columns

Format of Oracle Triggers

```
CREATE [OR REPLACE] TRIGGER TriggerName  
TriggerTiming TriggerEvent  
[ Referencing clause ]  
[ FOR EACH ROW ]  
[ WHEN ( Condition ) ]  
[ DECLARE sequence of declarative statements ]  
BEGIN sequence of statements  
[ EXCEPTION exception handling statements ]  
END;
```

Triggers



AFTER ROW Trigger Example

```
CREATE OR REPLACE TRIGGER tr_Enrollment_IA  
AFTER INSERT ON Enrollment
```

```
FOR EACH ROW
```

```
BEGIN
```

```
UPDATE Offering
```

```
SET OffNumEnrolled = OffNumEnrolled + 1  
WHERE OfferNo = :NEW.OfferNo;
```

```
EXCEPTION
```

```
WHEN OTHERS THEN
```

```
RAISE_Application_Error(-20001,'Database  
error');
```

```
END;
```

:NEW is a new record

Guide to Trigger Examples

❑ BEFORE ROW:

- Complex integrity constraints
- Transition constraints
- Standardization of data

❑ AFTER ROW

- Update propagation
- Audit trail
- Exception reporting

Compound Events in Triggers

- Compound events
 - Use OR to specify multiple events
 - Trigger body can detect the event
- Multiple triggers versus compound event triggers
 - More triggers but less complex
 - Fewer, more complex triggers
 - Trigger interaction increases with the number of triggers
 - No clear preference

Trigger Execution Procedure

- Inference engine that controls trigger firing
- Specifies execution order among triggers, integrity constraints, and manipulation statements
- Trigger body execution can cause other triggers to fire (*enrollment*→*offering*)
- SQL: standard trigger execution procedure
- Most DBMSs deviate from the standard

Simplified Oracle Trigger Execution Procedure

1. Execute the applicable **BEFORE STATEMENT** triggers.
2. For each row affected by the SQL manipulation statement:
 - 2.1 Execute the applicable **BEFORE ROW** triggers.
 - 2.2 Perform the data manipulation operation on the row.
 - 2.3 Perform integrity constraint checking.
 - 2.4 Execute the applicable **AFTER ROW** triggers.
3. Perform **deferred** integrity constraint checking.
4. Execute the applicable **AFTER statement** triggers.

Cascading trigger

SQL Statement

```
UPDATE t1 SET ...;
```

Fires the
UPDATE_T1
Trigger

UPDATE_T1 Trigger

```
BEFORE UPDATE ON t1
FOR EACH ROW
BEGIN
    .
    .
    .
    INSERT INTO t2 VALUES (...);
    .
    .
    .
END;
```

Fires the
INSERT_T2
Trigger

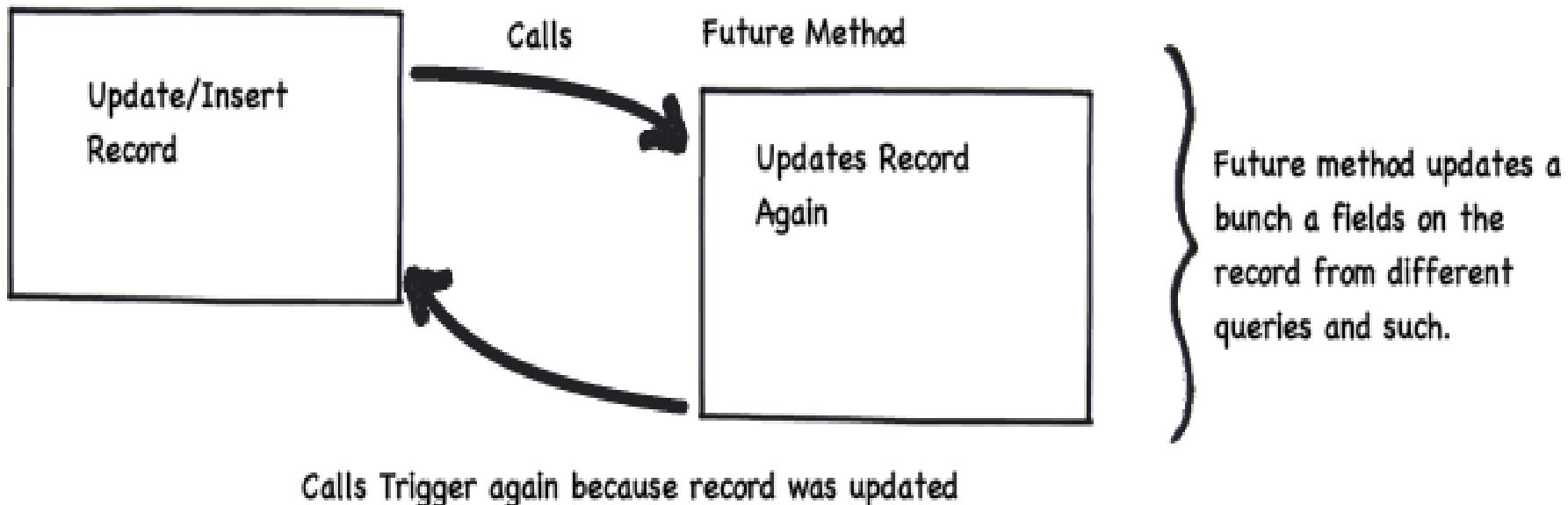
INSERT_T2 Trigger

```
BEFORE INSERT ON t2
FOR EACH ROW
BEGIN
    .
    .
    .
    INSERT INTO ... VALUES (...);
    .
    .
    .
END;
```

etc.

Recursive Trigger

Insert/Update Trigger



Overlapping Triggers

- ❑ Definition:
 - Two or more triggers with the same timing, granularity, and applicable event
 - Same SQL statement causes both triggers to fire
- ❑ SQL:2003 firing order based on trigger creation time
- ❑ Oracle: arbitrary firing order
- ❑ Carefully analyze overlapping triggers

Controlling Trigger Complexity

- ❑ Avoid data manipulation statements in BEFORE triggers
- ❑ Limit data manipulation statements in AFTER triggers.
- ❑ For triggers that fire on UPDATE statements, always list the columns.
- ❑ Ensure that overlapping triggers do not depend on a specific order to fire.

Summary

- Stored procedures and triggers are important for database application development and database administration
- Benefits for DBMS management of stored procedures
- Classification of triggers by granularity, timing, event, and purpose
- Knowledge of trigger execution procedures (depend on a DBMS)