PL-SQL Programming

Sanjay Patwardhan

Agenda

- Database concepts
- Introduction to programming blocks
- Basic programming
- Q&A

- Advance programming
- Q&A

- Object handling
- Exception handling
- Q&A

DataBase Concepts

Database Concepts

- **Interrelated data**
- **Set of programs to access that data**

Advantages :

- – **Controlling redundancy**
- – **Sharing data**
- – **Restricting unauthorized data**
- – **Providing multiple interfaces**
- – **Enforcing data constraints**
- – **Providing backup and recovery**
- – **Availability of updated information**

Three Levels Of Abstraction

Physical Level

Lowest level

Describes how the database is actually stored

Conceptual Level

Next higher level

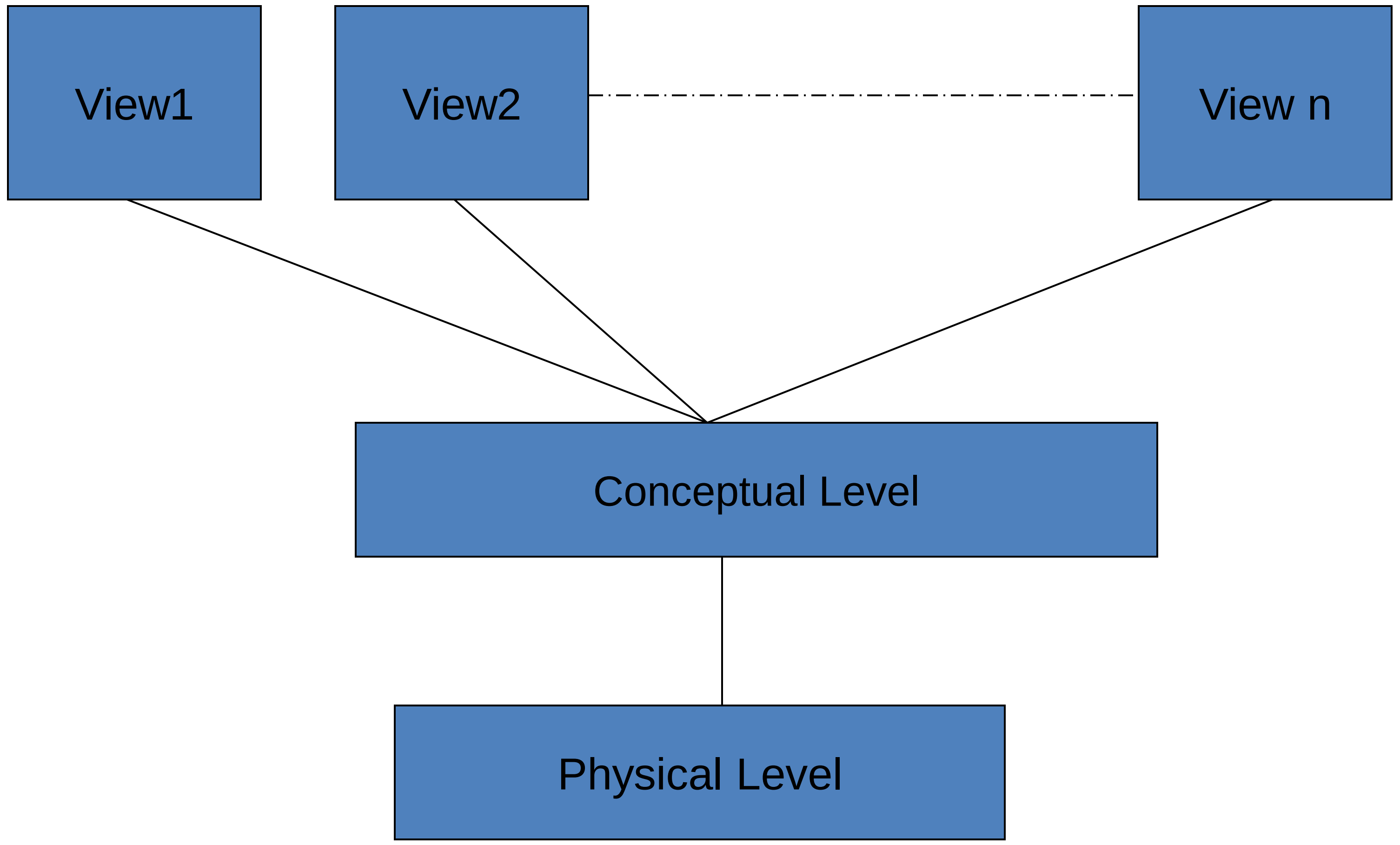
It describes what data are stored and the relationship between data

View Level

Highest level

It describes only the part of entire database

Three Levels Of Abstraction



Data Models

- **Object based logical model**
  - **E-R diagram**
- **Record based logical model**
  - **Hierarchical Model**
  - **Relational Model**
  - **Network Model**
- **Multi Dimensional model**
  - **ODM model (Object Dimension Model)**

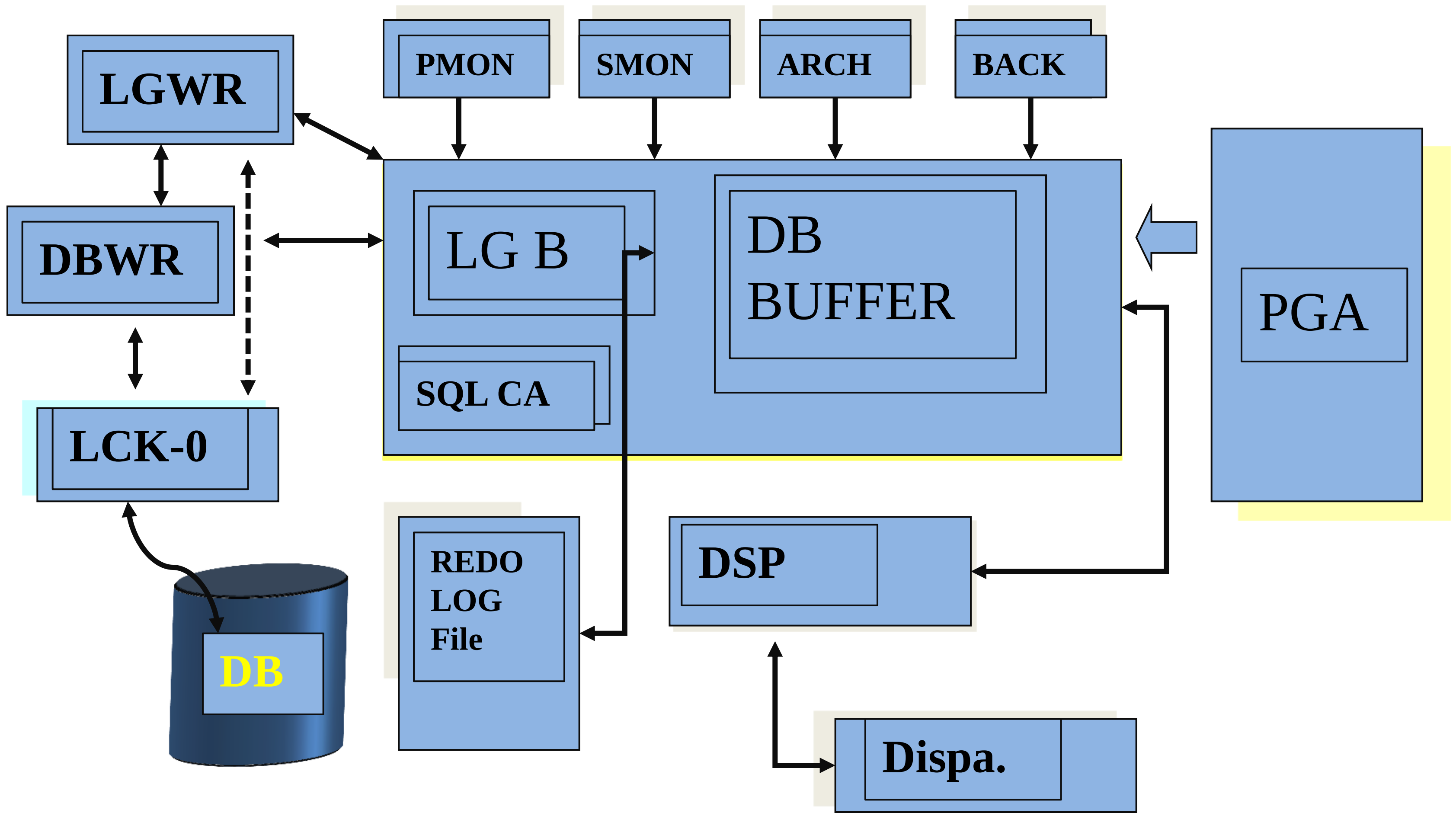
Database Administrator

- Scheme definition
- Granting of authorization for data access
- Integrity constraint specification

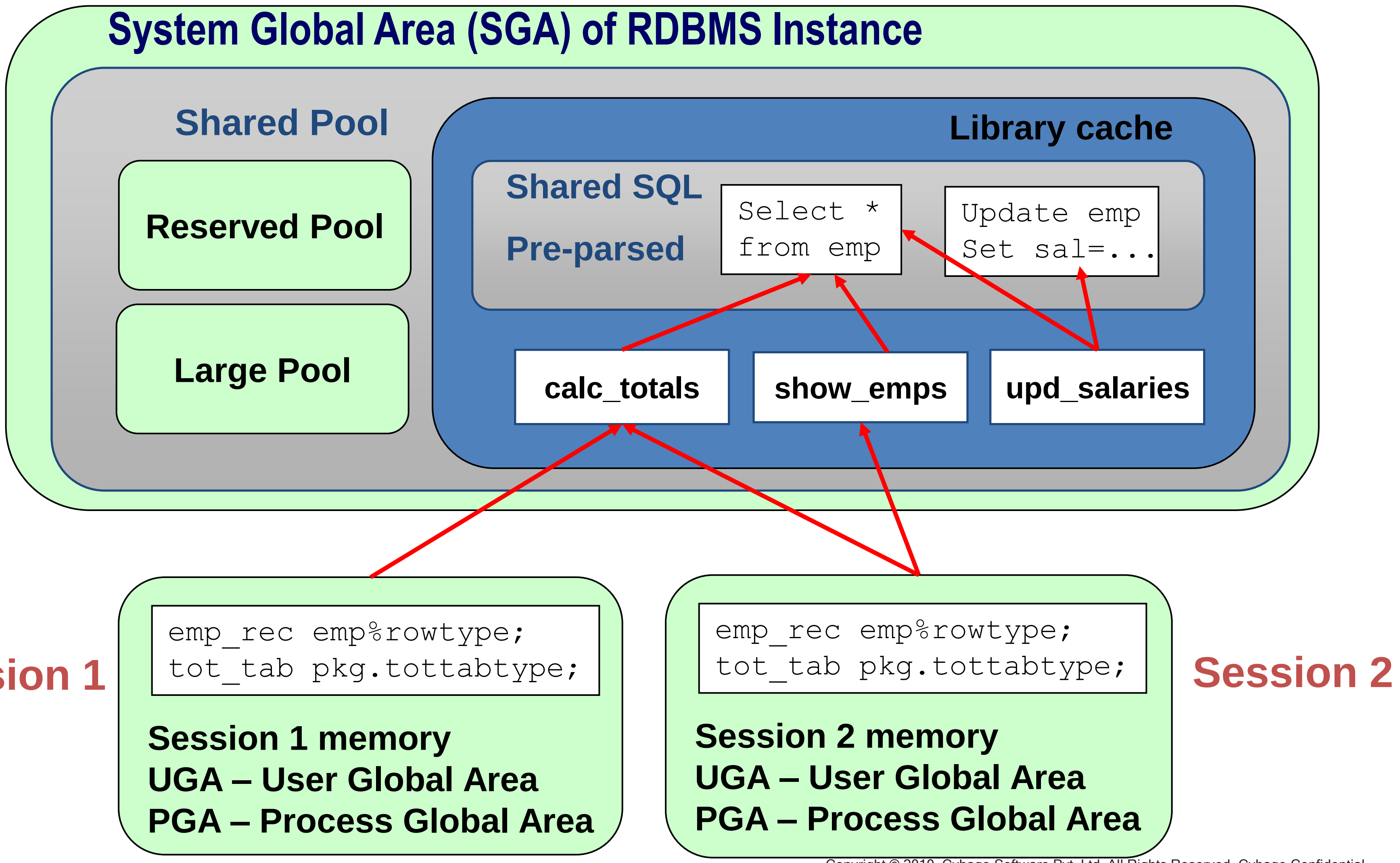
Database Manager

- Integrity enforcement
- Security enforcement
- Backup and recovery
- Concurrency control

RDBMS Engine



PL/SQL Runtime Memory Architecture



```
emp_rec emp%rowtype;
tot_tab pkg.tottabtype;
```

Session 1 memory
UGA – User Global Area
PGA – Process Global Area

```
emp_rec emp%rowtype;
tot_tab pkg.tottabtype;
```

Session 2 memory
UGA – User Global Area
PGA – Process Global Area

Questions…

Introduction to PLSQL

What is
PL/SQL?

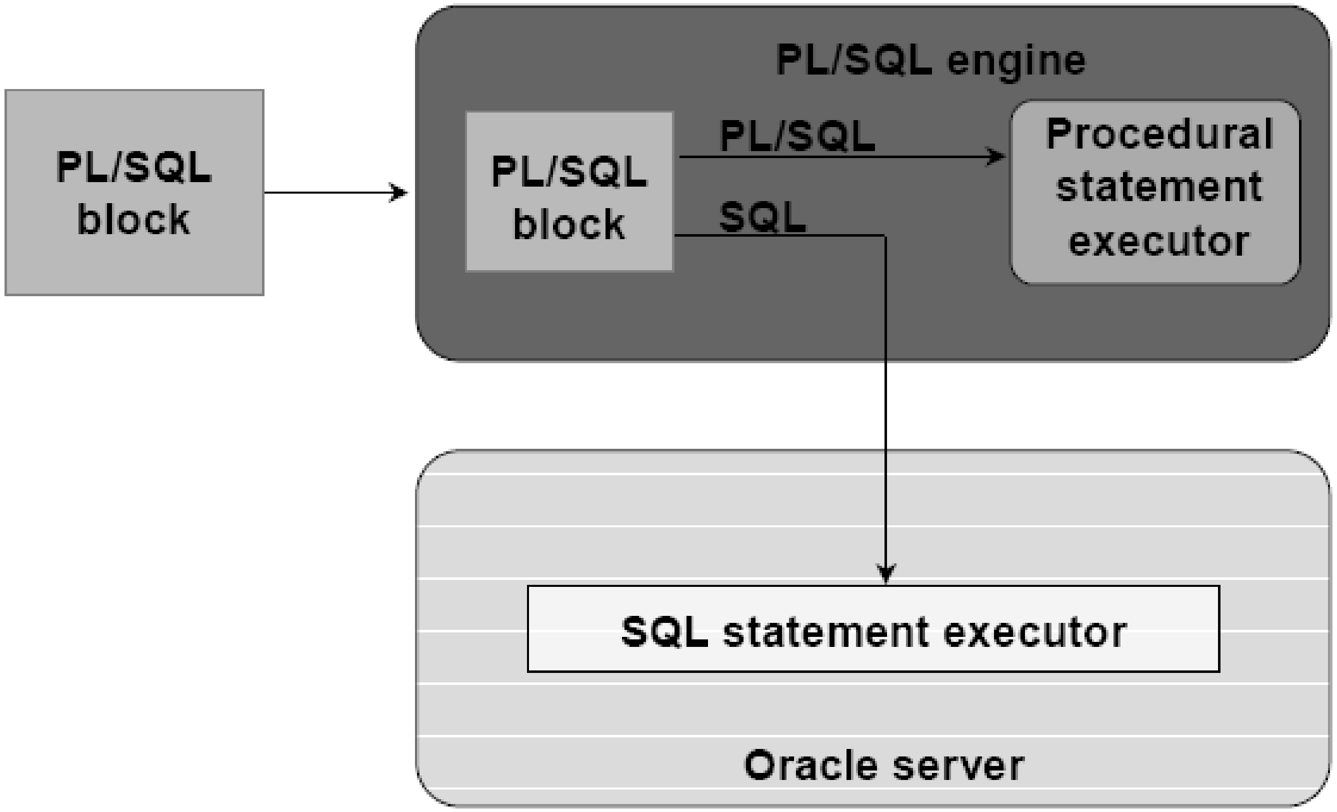
Procedural Language extension to SQL

About PL/SQL

- PL/SQL is the procedural extension to SQL with design features of programming languages.

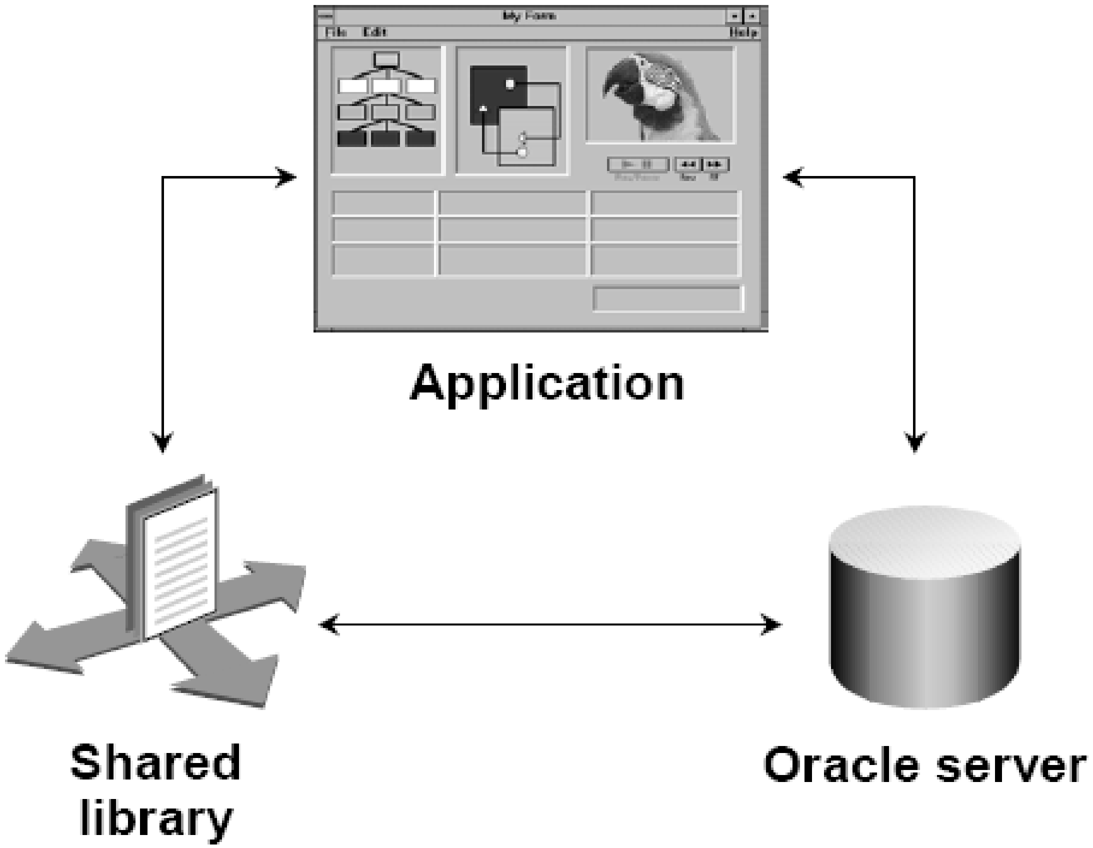
- Data manipulation and query statements of SQL are included within procedural units of code.

PL/SQL Environment



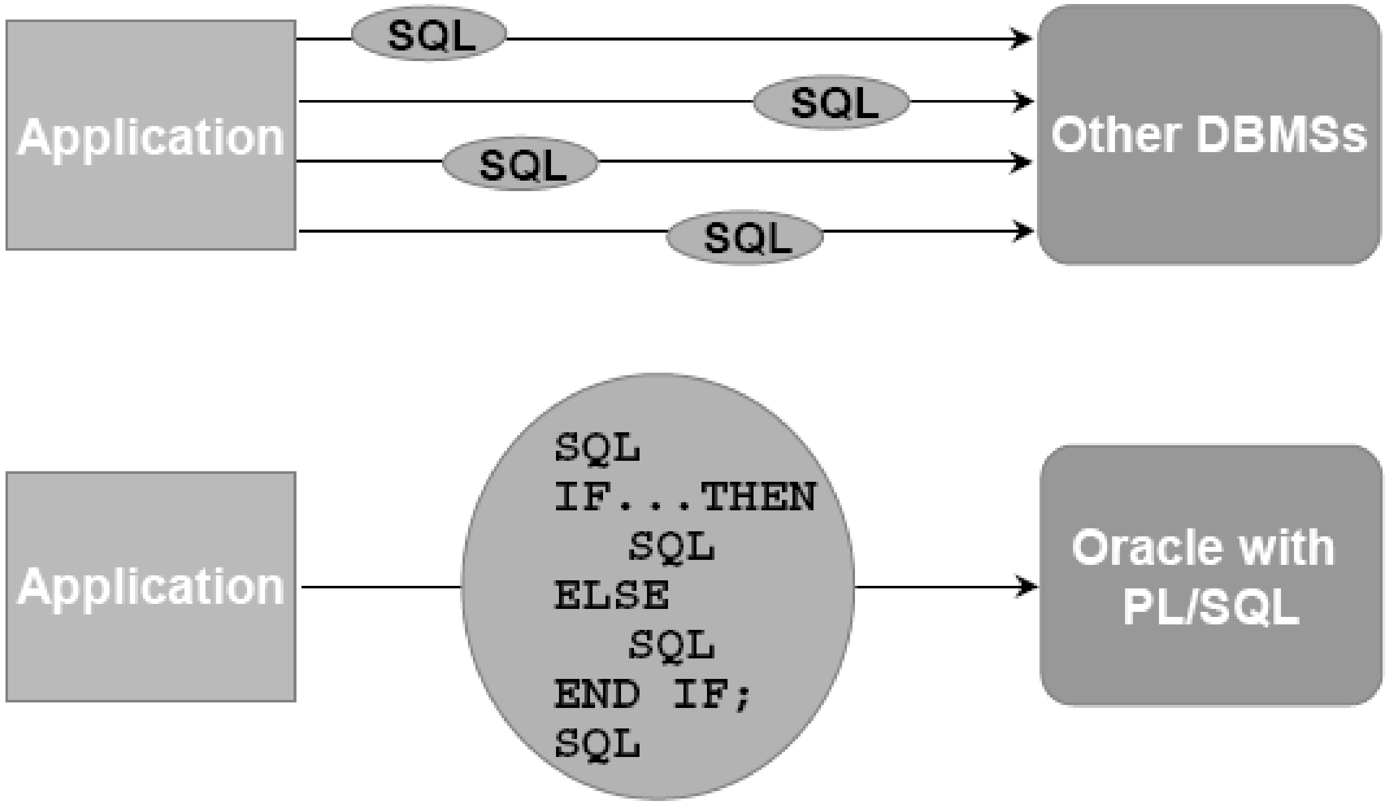
Benefits of PL/SQL

Integration



Benefits of PL/SQL

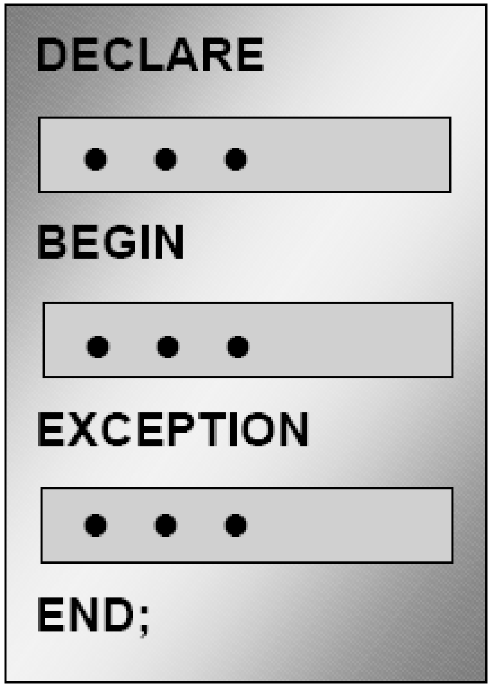
Improved performance



```
SQL
IF...THEN
    SQL
ELSE
    SQL
END IF;
SQL
```

Benefits of PL/SQL

- **Modularize program development**



Benefits of PL/SQL

- PL/SQL is Object Oriented.
- PL/SQL is portable.
- You can declare variables.
- You can program with procedural language control structures.
- PL/SQL can handle errors.

Benefits of Subprograms

- Easy maintenance
- Improved data security and integrity
- Improved performance
- Improved code clarity
- Reusability

Summary

PL/SQL is an extension to SQL.

Blocks of PL/SQL code are passed to and processed by a PL/SQL engine.

Benefits of PL/SQL:

– Integration

– Improved performance

– Portability

– Modularity of program development

Subprograms are named PL/SQL blocks, declared as either procedures or functions.

You can invoke subprograms from different environments.

What is required to build a program?

Structure
Declaring Variables
Process statements
Output

Objectives

After completing this Section, you should be able to :

- Recognize the basic PL/SQL block and its sections
- Describe the significance of variables in PL/SQL
- Declare PL/SQL variables
- Execute a PL/SQL block

PL/SQL

- Allows using general programming tools with SQL, for example: loops, conditions, functions, etc.
- This allows a lot more freedom than general SQL, and is lighter-weight than JDBC.
- We write PL/SQL code in a regular file, for example PL.sql, and load it with @PL in the sqlplus console.

PL/SQL Blocks

PL/SQL code is built of Blocks, with a unique structure.

There are two types of blocks in PL/SQL:

1. **Anonymous Blocks:** have no name (like scripts)

   can be written and executed immediately in SQLPLUS

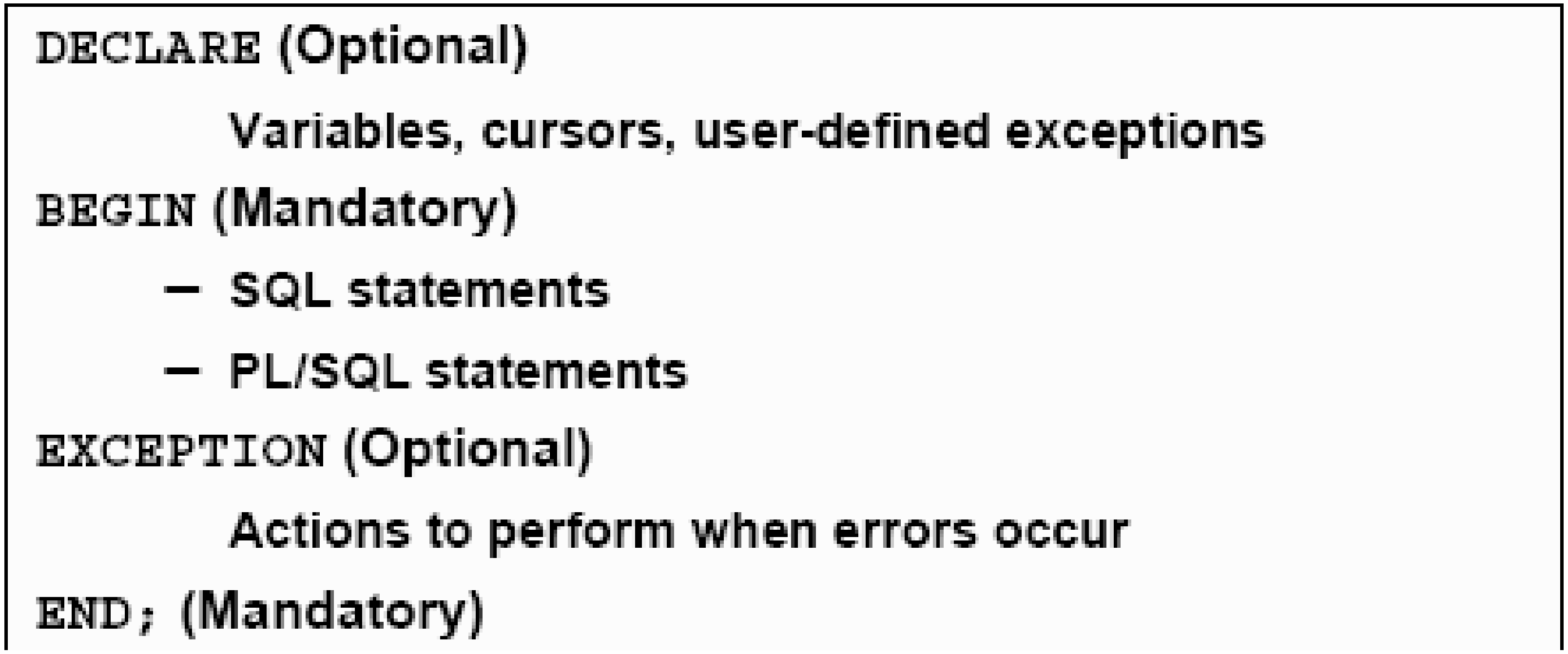
   can be used in a trigger

2. **Named Blocks:**

   Procedures

   Functions

PL/SQL Block Structure



DECLARE (Optional)

    Variables, cursors, user-defined exceptions

BEGIN (Mandatory)

- SQL statements
- PL/SQL statements

EXCEPTION (Optional)

    Actions to perform when errors occur

END; (Mandatory)

Anonymous Block Structure:

DECLARE (optional)

/* Here you declare the variables you will use in this block */

BEGIN (mandatory)

/* Here you define the executable statements (what the block DOES!)*/

EXCEPTION (optional)

/* Here you define the actions that take place if an exception is thrown during the run of this block */

END; (mandatory)

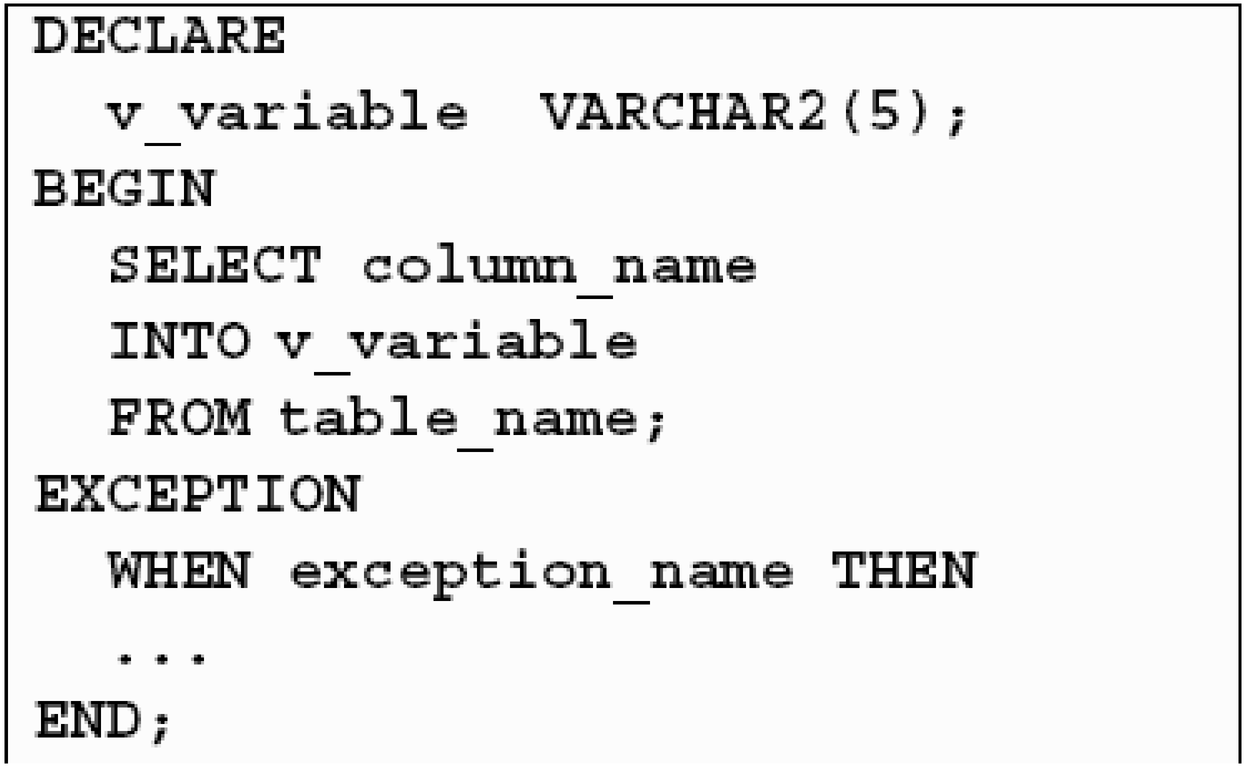
/

Always put a new line with only a / at the end of a block! (This tells Oracle to run the block)

A correct completion of a block will generate the following message:

PL/SQL procedure successfully completed

Executing Statements and PL/SQL Blocks



```
DECLARE
  v_variable  VARCHAR2(5);
BEGIN
  SELECT column_name
  INTO v_variable
  FROM table_name;
EXCEPTION
  WHEN exception_name THEN
  ...
END;
```

Block Types

Anonymous

```
[DECLARE]


BEGIN
  --statements

[EXCEPTION]

END;
```

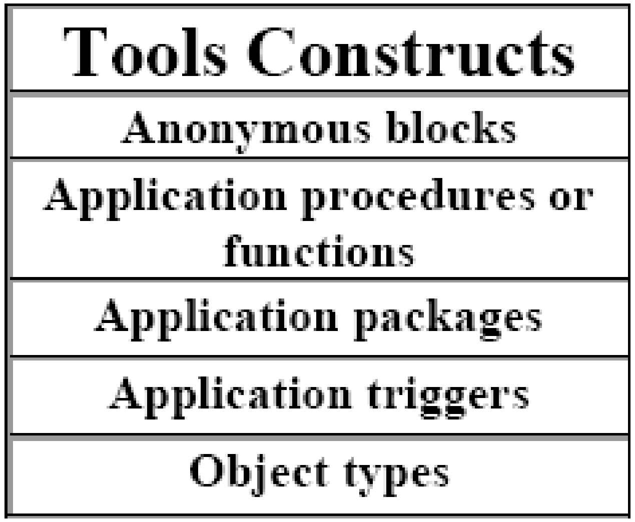
Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

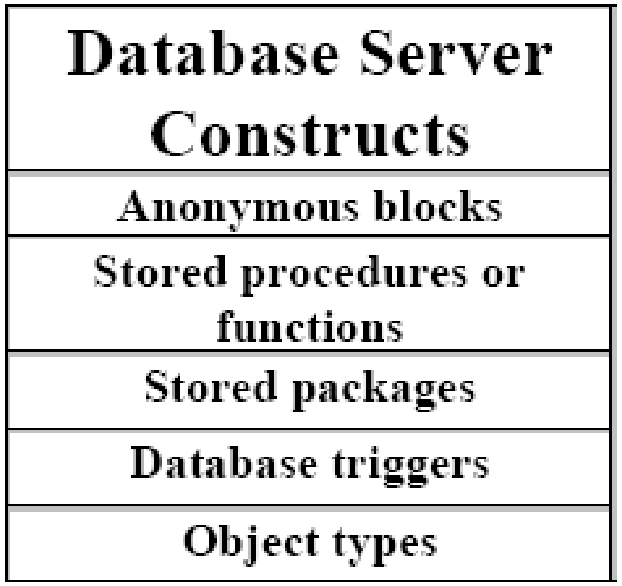
Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

Program Constructs



Tools Constructs
Anonymous blocks
Application procedures or functions
Application packages
Application triggers
Object types



Database Server Constructs
Anonymous blocks
Stored procedures or functions
Stored packages
Database triggers
Object types

Use of Variables

- Scope of Variables
  - Global
  - Local
- Variables can be used for:
  - Temporary storage of data
  - Manipulation of stored values
  - Reusability
  - Ease of maintenance

Handling Variables in PL/SQL

- Declare and initialize variables in the declaration section.
- Assign new values to variables in the executable section.
- Pass values into PL/SQL blocks through parameters.
- View results through output variables.

Types of Variables

- PL/SQL variables:

- Scalar
- Composite
- Reference
- LOB (large objects)

- Non-PL/SQL variables: Bind and host variables

Using iSQL*Plus Variables Within PL/SQL Blocks

- **PL/SQL does not have input or output capability of its own.**

- **You can reference substitution variables within a PL/SQL block with a preceding ampersand.**

- ***i*SQL*Plus host (or “bind”) variables can be used to pass run time values out of the PL/SQL block back to the *i*SQL*Plus environment.**

Types of Variables

TRUE

25-JAN-01

256120.08

"Four score and seven years ago our fathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."

Atlanta

Declaring PL/SQL Variables

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
      [:= | DEFAULT expr];
```

Examples:

```
DECLARE
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

Guidelines for Declaring PL/SQL Variables

- Follow naming conventions.

- Initialize variables designated as NOT NULL and CONSTANT.

- Declare one identifier per line.

- Initialize identifiers by using the assignment operator (:=) or the DEFAULT reserved word.

identifier := *expr*;

Naming Rules

- Two variables can have the same name, provided they are in different blocks.

- The variable name (identifier) should not be the same as the name of table columns used in the block.

```
DECLARE
  employee_id  NUMBER(6);
BEGIN
  SELECT     employee_id
  INTO       employee_id
  FROM       employees
  WHERE      last_name = 'Kochhar';
END;
/
```

Adopt a naming convention for PL/SQL identifiers: for example, v_employee_id

Variable Initialization and Keywords

Assignment operator (:=)

DEFAULT keyword

NOT NULL constraint

Syntax:

```
identifier := expr;
```

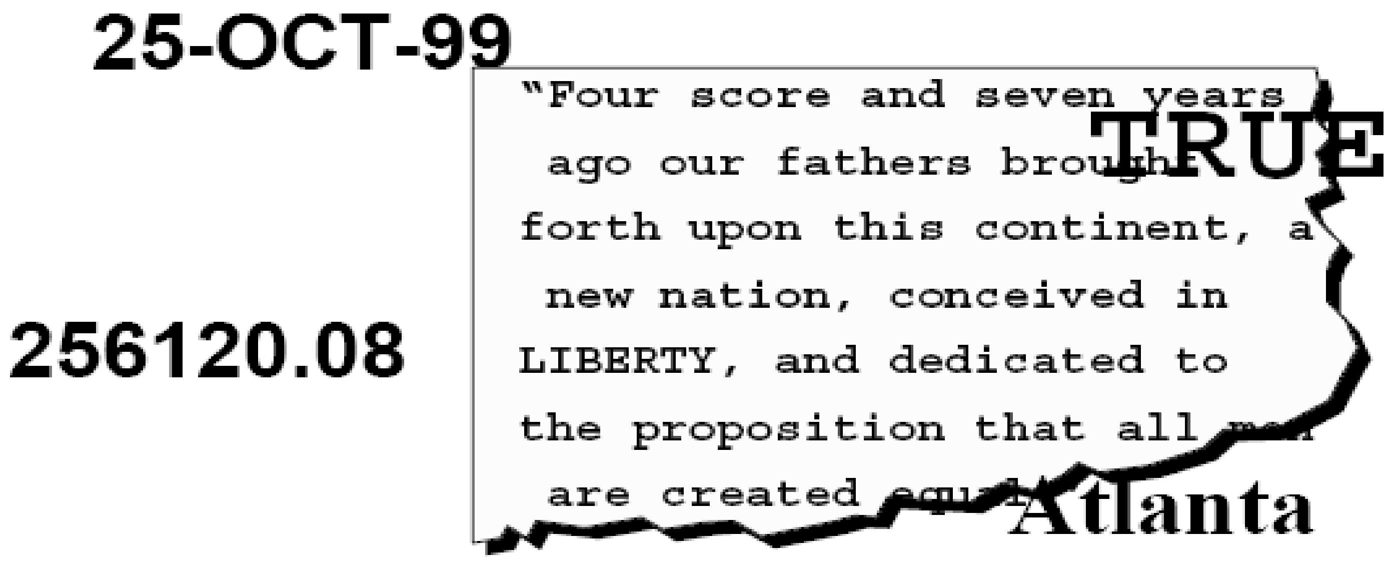
Examples:

```
v_hiredate := '01-JAN-2001';
```

```
v_ename := 'Maduro';
```

Scalar Data Types

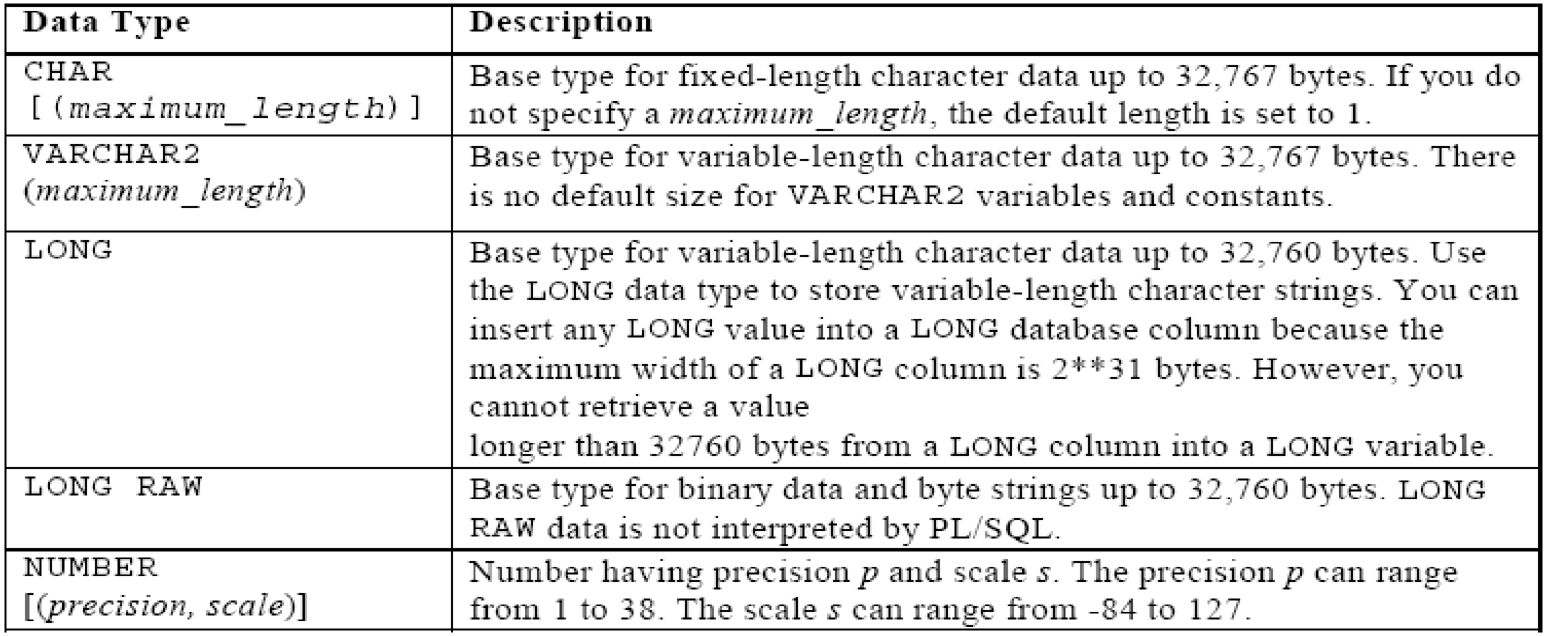
- Hold a single value
- Have no internal components


25-OCT-99
"Four score and seven years ago our fathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."
TRUE
256120.08
Atlanta

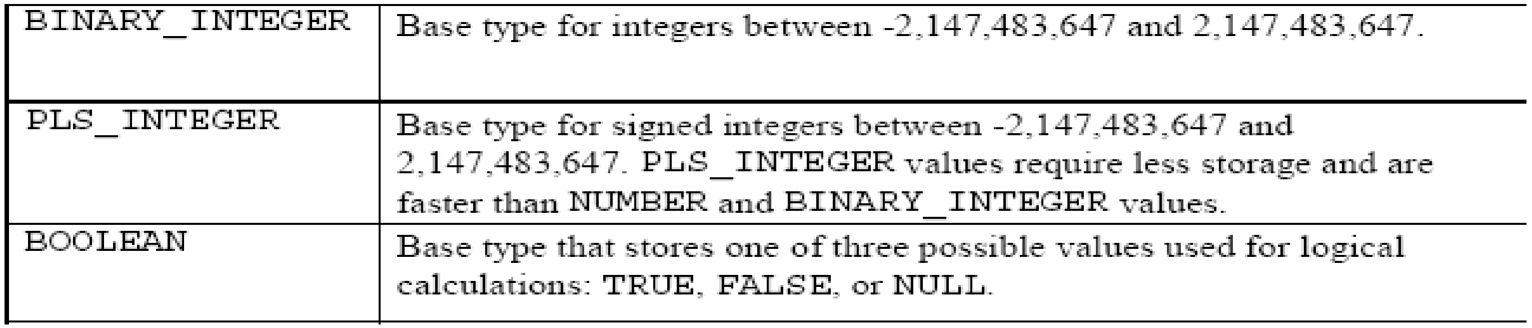
Base Scalar Data Types

- CHAR [(*maximum_length*)]
- VARCHAR2 (*maximum_length*)
- LONG
- LONG RAW
- NUMBER [(*precision*, *scale*)]
- BINARY_INTEGER
- PLS_INTEGER
- BOOLEAN
- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Base Scalar Data Types



Data Type	Description
CHAR [(*maximum_length*)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a *maximum_length*, the default length is set to 1.
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
LONG	Base type for variable-length character data up to 32,760 bytes. Use the LONG data type to store variable-length character strings. You can insert any LONG value into a LONG database column because the maximum width of a LONG column is 2**31 bytes. However, you cannot retrieve a value longer than 32760 bytes from a LONG column into a LONG variable.
LONG RAW	Base type for binary data and byte strings up to 32,760 bytes. LONG RAW data is not interpreted by PL/SQL.
NUMBER [(*precision*, *scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from -84 to 127.



BINARY_INTEGER	Base type for integers between -2,147,483,647 and 2,147,483,647.
PLS_INTEGER	Base type for signed integers between -2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER and BINARY_INTEGER values.
BOOLEAN	Base type that stores one of three possible values used for logical calculations: TRUE, FALSE, or NULL.

LARGE OBJECT (LOB) DATATYPES

Data Type Syntax	Oracle 9i	Oracle 10g	Oracle 11g	Explanation (if applicable)
bfile	Maximum file size of 4GB.	Maximum file size of 2^{32}-1 bytes.	Maximum file size of 2^{64}-1 bytes.	File locators that point to a binary file on the server file system (outside the database).
blob	Store up to 4GB of binary data.	Store up to (4 gigabytes -1) * (the value of the CHUNK parameter of LOB storage).	Store up to (4 gigabytes -1) * (the value of the CHUNK parameter of LOB storage).	Stores unstructured binary large objects.
clob	Store up to 4GB of character data.	Store up to (4 gigabytes -1) * (the value of the CHUNK parameter of LOB storage) of character data.	Store up to (4 gigabytes -1) * (the value of the CHUNK parameter of LOB storage) of character data.	Stores single-byte and multi-byte character data.
nclob	Store up to 4GB of character text data.	Store up to (4 gigabytes -1) * (the value of the CHUNK parameter of LOB storage) of character text data.	Store up to (4 gigabytes -1) * (the value of the CHUNK parameter of LOB storage) of character text data.	Stores unicode data.

New data types

The new data types available in Oracle 11g are:

Binary XML type - up to 15 x faster over XML LOBs.
New "SIMPLE_INTEGER" data type - always NOT NULL
DICOM Medical Images.
3D Spatial Support.
RFID tag data types.

New CONTINUE statement - starts the next iteration of the loop
Ability to reference sequences (no need to select seq.nextval into :var from dual)

Important PL/SQL delimiters

+, **-**, *****, **/**	arithmetic operators
;	statement terminator
:=	assignment operator
=>	association operator
||	strings concatenation operator
.	component indicator
%	attribute operator
‘	character string delimiter
--	single line comment
/*, ***/**	multi line comment delimiters
..	range operator
=, **>**, **>=**, **<**, **<=**	relational operators
!=, **~=**, **^=**, **<>**	not equal relational operators
is null, like, between	PL/SQL relational operators

Scalar Variable Declarations

Examples:

```
DECLARE
   v_job          VARCHAR2(9);
   v_count        BINARY_INTEGER := 0;
   v_total_sal    NUMBER(9,2)  := 0;
   v_orderdate    DATE := SYSDATE + 7;
   c_tax_rate     CONSTANT NUMBER(3,2)  := 8.25;
   v_valid        BOOLEAN NOT NULL := TRUE;
   ...
```

DECLARE

Syntax

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

Examples

Notice that PL/SQL includes all SQL types, and more…

```
Declare
  birthday      DATE;
  age             NUMBER(2) NOT NULL := 27;
  name           VARCHAR2(13) := 'Levi';
  magic         CONSTANT NUMBER := 77;
  valid         BOOLEAN NOT NULL := TRUE;
```

Declaring Variables with the %TYPE Attribute

Examples

Accessing column sname in table Sailors

```
DECLARE
   sname                        Sailors.sname%TYPE;
   fav_boat                     VARCHAR2(30);
  my_fav_boat                   fav_boat%TYPE :=
'Pinta';
...
```

Accessing another variable

Declaring with the %ROWTYPE Attribute

Declare a variable with the type of a ROW of a table.

Accessing table Reserves

```
reserves_record         Reserves%ROWTYPE;
```

And how do we access the fields in reserves_record?

```
reserves_record.sid:=9;
Reserves_record.bid:=877;
```

Creating a PL/SQL Record

A record is a type of variable which we can define (like ‘struct’ in C or ‘object’ in Java)

```
DECLARE
  TYPE sailor_record_type IS RECORD
    (sname          VARCHAR2(10),
     sid            VARCHAR2(9),
     age         NUMBER(3),
     rating       NUMBER(3));
  sailor_record       sailor_record_type;
...
BEGIN
  Sailor_record.sname:=‘peter’;
  Sailor_record.age:=45;
…
```

Declaring Boolean Variables

Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.

- The variables are compared by the logical operators AND, OR, and NOT.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

Composite Data Types

TRUE	23-DEC-98	ATLANTA	

PL/SQL table structure

1	SMITH
2	JONES
3	NANCY
4	TIM

VARCHAR2

BINARY_INTEGER

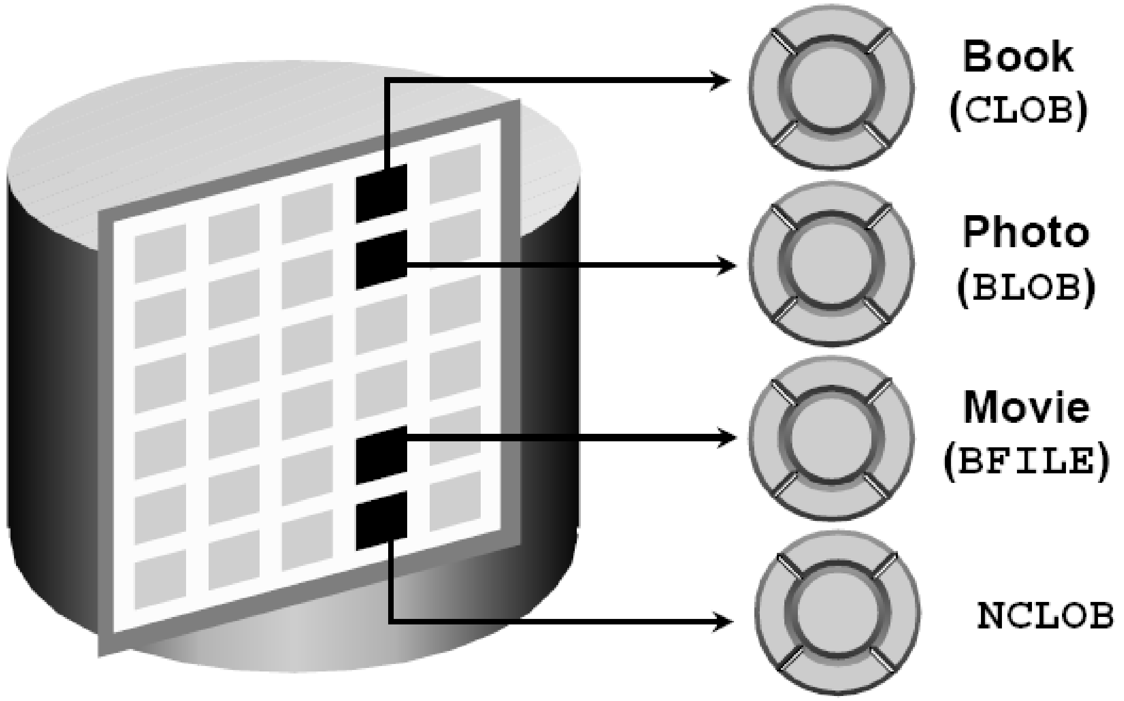
PL/SQL table structure

1	5000
2	2345
3	12
4	3456

NUMBER

BINARY_INTEGER

LOB Data Type Variables



Using Bind Variables

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).

Example:

```
VARIABLE      g_salary NUMBER
BEGIN
    SELECT    salary
    INTO      :g_salary
    FROM      employees
    WHERE     employee_id = 178;
END;
/
PRINT g_salary
```

Referencing Non-PL/SQL Variables

Store the annual salary into a *i*SQL*Plus host variable.

- **Reference non-PL/SQL variables as host variables.**
- **Prefix the references with a colon (:).**

  :g_monthly_sal := v_sal / 12;

```
VARIABLE g_monthly_sal NUMBER
DEFINE p_annual_sal = 50000
SET VERIFY OFF
DECLARE
v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
:g_monthly_sal := v_sal/12;
END;
/
PRINT g_monthly_sal
```

DBMS_OUTPUT.PUT_LINE

- **An Oracle-supplied packaged procedure**
- **An alternative for displaying data from a PL/SQL block**
- **Must be enabled in *i*SQL*Plus with SET SERVEROUTPUT ON**

```
SET SERVEROUTPUT ON
DEFINE p_annual_sal = 60000
```

```
DECLARE
  v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
  v_sal := v_sal/12;
  DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' ||
                        TO_CHAR(v_sal));
END;
/
```

Summary

In this Section you should have learned that:

- **PL/SQL blocks are composed of the following sections:**
  - **Declarative (optional)**
  - **Executable (required)**
  - **Exception handling (optional)**

- **A PL/SQL block can be an anonymous block, procedure, or function.**

Summary

In this Section you should have learned that:

- **PL/SQL identifiers:**
  - **Are defined in the declarative section**
  - **Can be of scalar, composite, reference, or LOB data type**
  - **Can be based on the structure of another variable or database object**
  - **Can be initialized**
- **Variables declared in an external environment such as *i*SQL*Plus are called host variables.**
- **Use DBMS_OUTPUT.PUT_LINE to display data from a PL/SQL block.**

Writing Control Structures

IF Statements

Syntax:

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

If the employee name is Gietz, set the Manager ID to 102.

```
IF UPPER(v_last_name) = 'GIETZ' THEN
  v_mgr := 102;
END IF;
```

Simple IF Statements

If the last name is Vargas:

- Set job ID to SA_REP
- Set department number to 80

```
. . .
IF v_ename      = 'Vargas' THEN
   v_job       := 'SA_REP';
   v_deptno    := 80;
END IF;
. . .
```

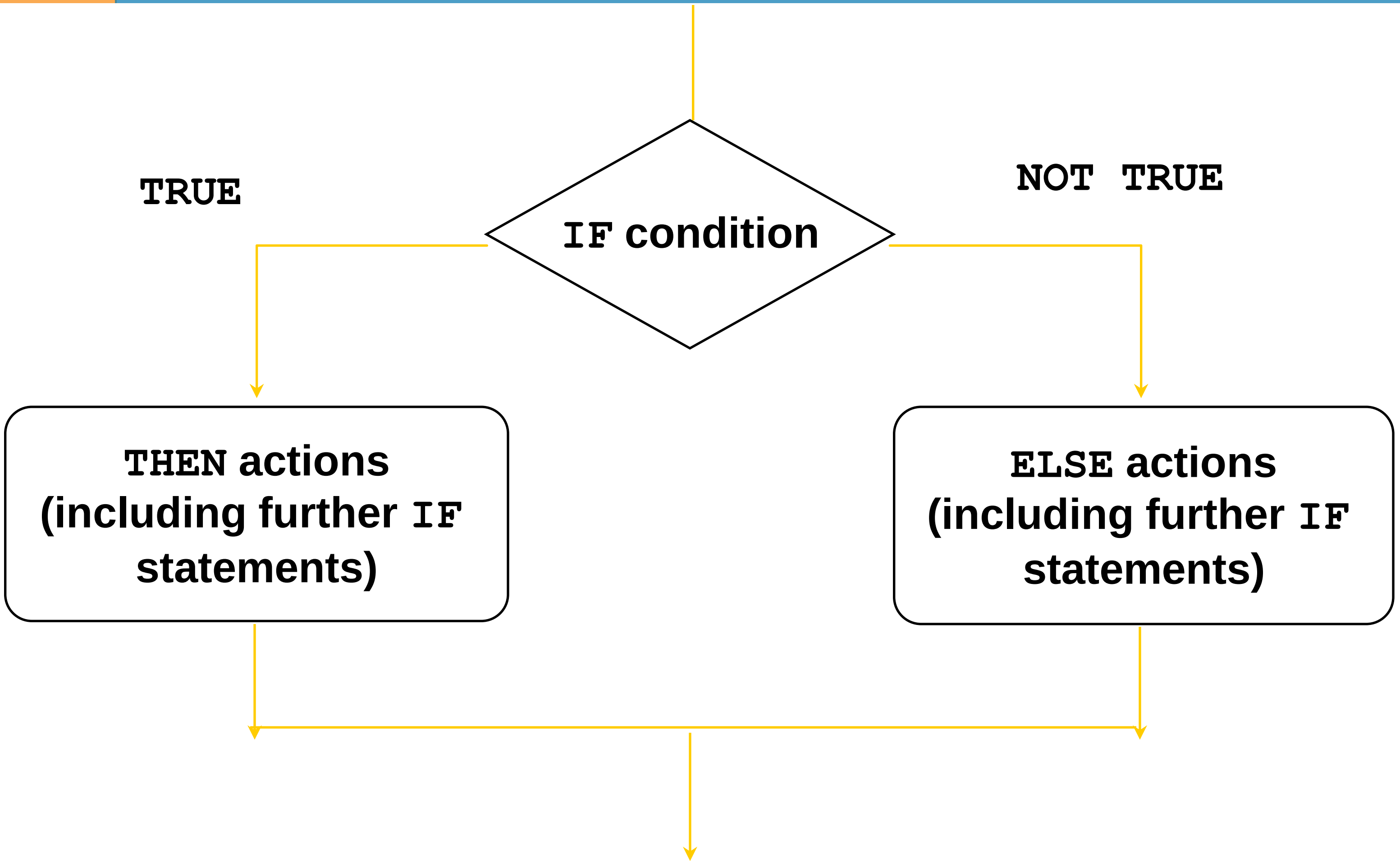
Compound IF Statements

If the last name is Vargas and the salary is more than 6500:

Set department number to 60.

```
. . .
IF v_ename  = 'Vargas' AND salary > 6500 THEN
   v_deptno := 60;
END IF;
. . .
```

IF-THEN-ELSE Statement Execution Flow

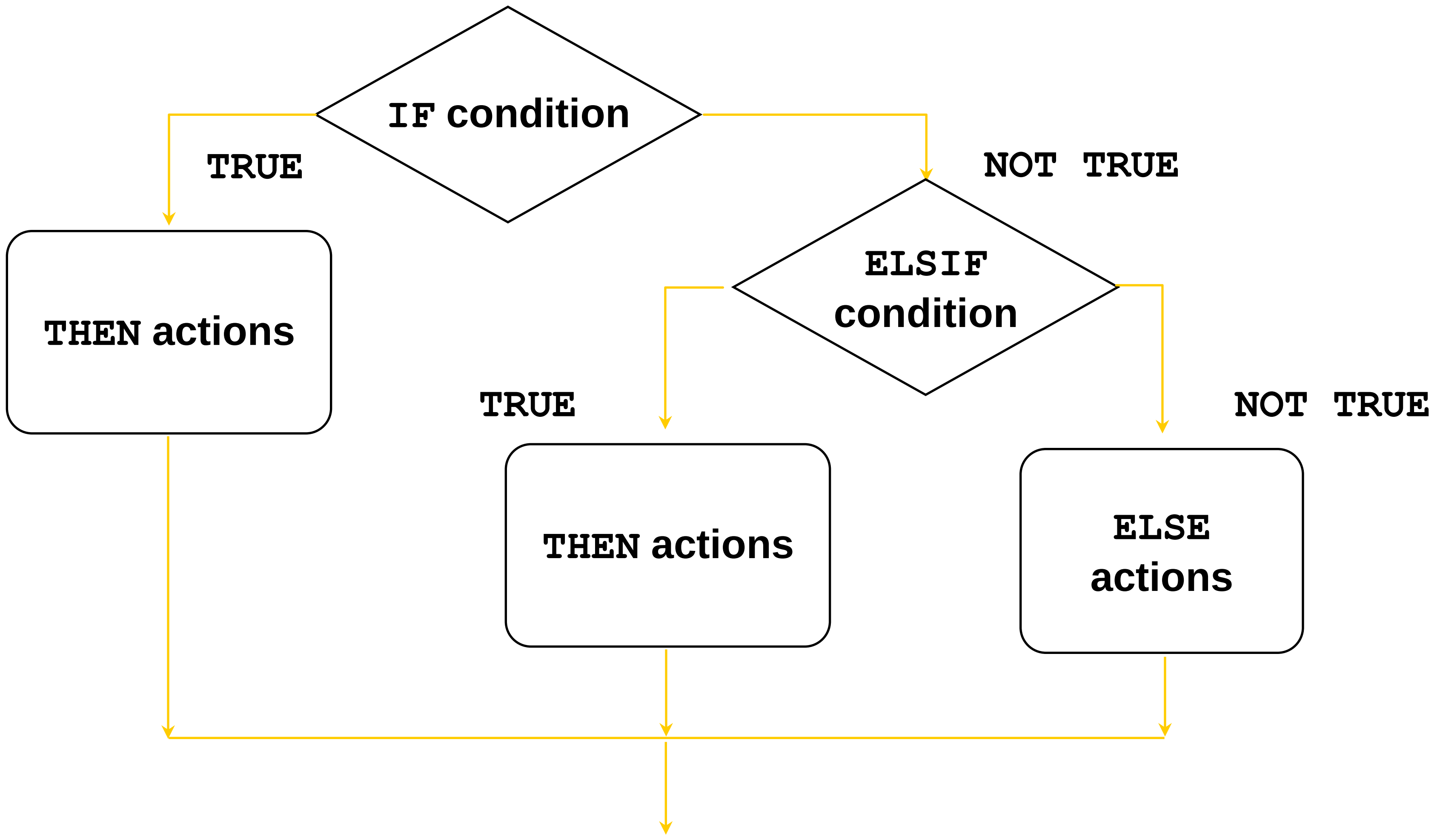

TRUE
NOT TRUE
IF condition
THEN actions (including further IF statements)
ELSE actions (including further IF statements)

IF-THEN-ELSE Statements

Set a Boolean flag to *TRUE* if the hire date is greater than five years; otherwise, set the Boolean flag to *FALSE*.

```
DECLARE
   v_hire_date  DATE := '12-Dec-1990';
   v_five_years BOOLEAN;
BEGIN
.  .  .
IF MONTHS_BETWEEN(SYSDATE,v_hire_date)/12 > 5 THEN
   v_five_years := TRUE;
ELSE
   v_five_years := FALSE;
END IF;
...
```

IF-THEN-ELSIF Statement Execution Flow



IF-THEN-ELSIF Statements

For a given value, calculate a percentage of that value based on a condition.

Example:

```
. . .
IF    v_start > 100 THEN
      v_start := 0.2 * v_start;
ELSIF v_start >= 50 THEN
      v_start := 0.5 * v_start;
ELSE
      v_start := 0.1 * v_start;
END IF;
. . .
```

CASE Expressions

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses an expression whose value is used to select one of several alternatives.

```
CASE selector
    WHEN expression1 THEN result1
    WHEN expression2 THEN result2
    ...
    WHEN expressionN THEN resultN
   [ELSE resultN+1;]
END;
```

CASE Expressions: Example

```
SET SERVEROUTPUT ON
DECLARE
   v_grade CHAR(1) := UPPER('&p_grade');
   v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal :=
      CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade || '
                       Appraisal ' || v_appraisal);
END;
/
```

Boolean Conditions

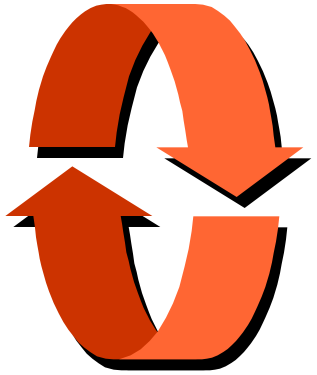
What is the value of `V_FLAG` in each case?

```
v_flag := v_reorder_flag AND v_available_flag;
```

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	?
TRUE	FALSE	?
NULL	TRUE	?
NULL	FALSE	?

Iterative Control: LOOP Statements

- Loops repeat a statement or sequence of statements multiple times.
- There are three loop types:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop



Basic Loops

Syntax:

```
LOOP                                          -- delimiter
   statement1;                                -- statements
   . . .
   EXIT [WHEN condition];                     -- EXIT statement
END LOOP;                                     -- delimiter
```

```
condition     is a Boolean variable or
              expression (TRUE, FALSE, or NULL);
```

Basic Loops

Example:

```
DECLARE
  v_country_id    locations.country_id%TYPE := 'CA';
  v_location_id   locations.location_id%TYPE;
  v_counter       NUMBER(2) := 1;
  v_city          locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_location_id FROM locations
  WHERE country_id = v_country_id;
  LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_location_id + v_counter),v_city, v_country_id);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 3;
  END LOOP;
END;
/
```

WHILE Loops

Syntax:

```
WHILE condition LOOP
   statement1;
   statement2;
   . . .
END LOOP;
```

Condition is evaluated at the beginning of each iteration.

WHILE Loops

```
DECLARE
  v_country_id      locations.country_id%TYPE := 'CA';
  v_location_id     locations.location_id%TYPE;
  v_city            locations.city%TYPE := 'Montreal';
  v_counter         NUMBER  := 1;
BEGIN
  SELECT MAX(location_id) INTO v_location_id FROM locations
  WHERE country_id = v_country_id;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_location_id + v_counter), v_city, v_country_id);
    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

FOR Loops

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- Use a `FOR` loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.
- '`lower_bound .. upper_bound`' is required syntax.

FOR Loops

Insert three new locations IDs for the country code of CA and the city of Montreal.

```
DECLARE
  v_country_id    locations.country_id%TYPE := 'CA';
  v_location_id   locations.location_id%TYPE;
  v_city          locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_location_id
    FROM locations
    WHERE country_id = v_country_id;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_location_id + i), v_city, v_country_id );
  END LOOP;
END;
/
```

Guidelines While Using Loops

- Use the basic loop when the statements inside the loop must execute at least once.
- Use the `WHILE` loop if the condition has to be evaluated at the start of each iteration.
- Use a `FOR` loop if the number of iterations is known.

Writing Explicit Cursors

Objectives

After completing this Section, you should be able to do the following:

Distinguish between an implicit and an explicit cursor

Discuss when and why to use an explicit cursor

Use a PL/SQL record variable

Write a cursor FOR loop

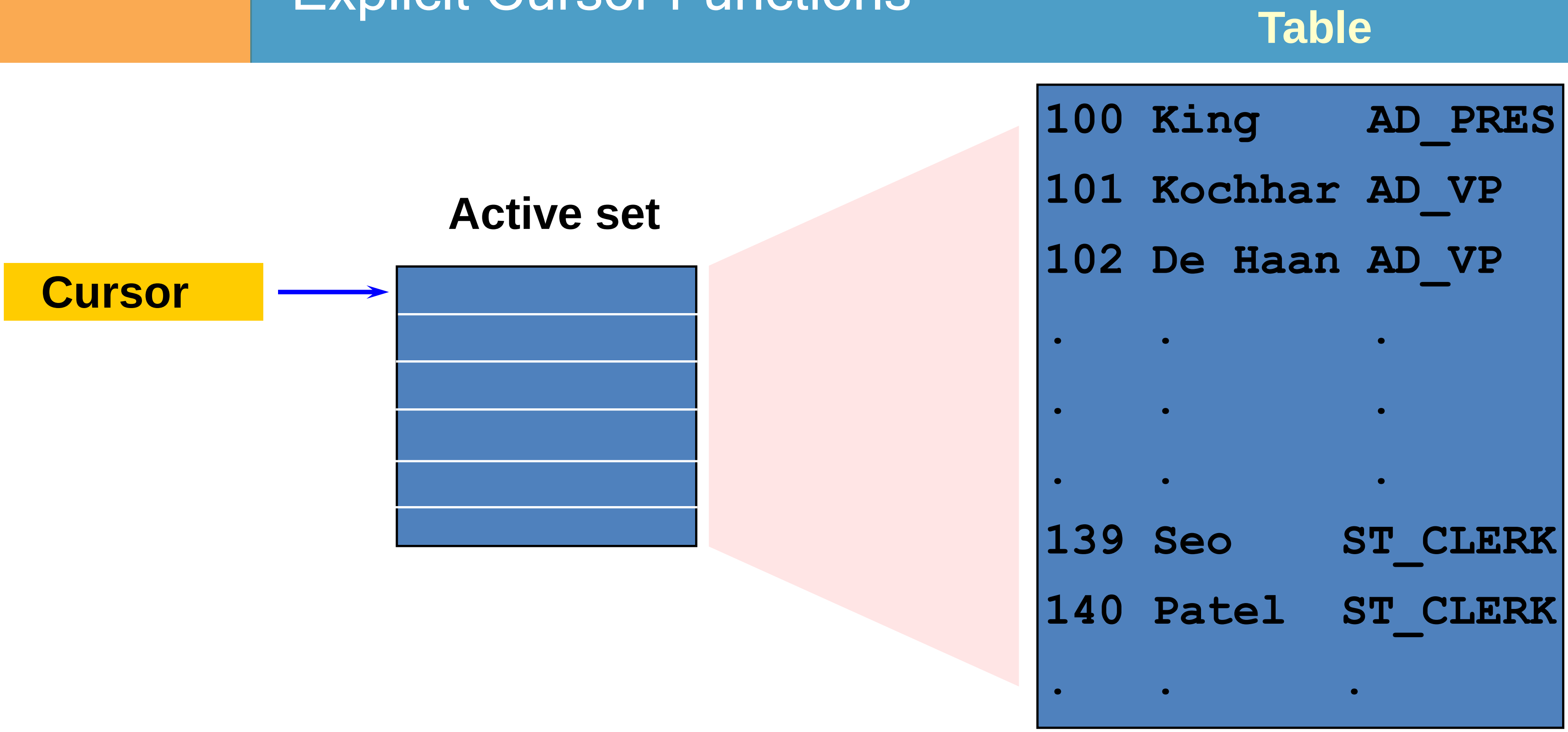
About Cursors

Every SQL statement executed by the Oracle Server
has an individual cursor associated with it:

Implicit cursors: Declared for all DML and PL/SQL `SELECT` statements

Explicit cursors: Declared and named by the programmer

Explicit Cursor Functions



Cursors

each SQL query produces a result set - cursor

- set of rows
- resides on the server in a session's process memory

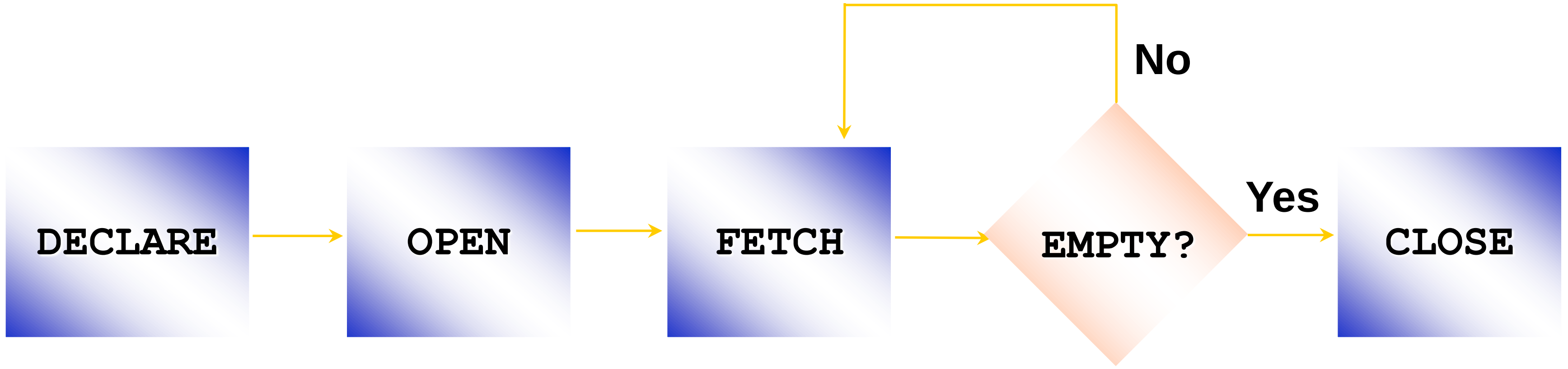
PL/SQL program can read the result set in iterating fashion

```
select
   emp_no
  ,emp_name
  ,emp_job
from employees
 where emp_no > 500;
```

Result Set

EMP_NO	EMP_NAME	EMP_JOB	EMP_HIREDATE	EMP_DEPTNO
380	KING	CLERK	1-JAN-1982	10
381	BLAKE	ANALYST	11-JAN-1982	30
392	CLARK	CLERK	1-FEB-1981	30
569	SMITH	CLERK	2-DEC-1980	20
566	JONES	MANAGER	5-JUL-1978	30
788	SCOTT	ANALYST	20-JUL-1981	10
876	ADAMS	CLERK	14-MAR-1980	10
902	FORD	ANALYST	25-SEP-1978	20

Controlling Explicit Cursors



- Create a named SQL area

- Identify the active set

- Load the current row into variables

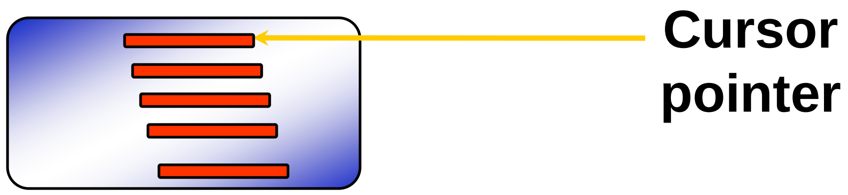
- Test for existing rows
- Return to `FETCH` if rows are found

- Release the active set

Controlling Explicit Cursors

1. Open the cursor

1. Open the cursor.

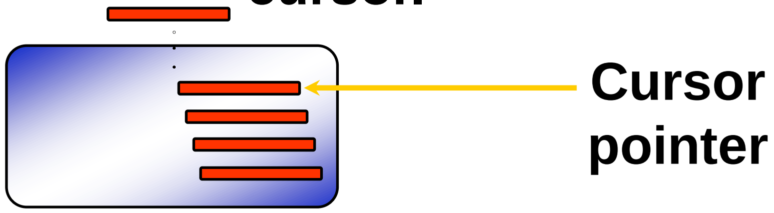


2. Fetch a row
3. Close the Cursor

Controlling Explicit Cursors

1. **Open the cursor**
2. **Fetch a row**

2. Fetch a row using the cursor.



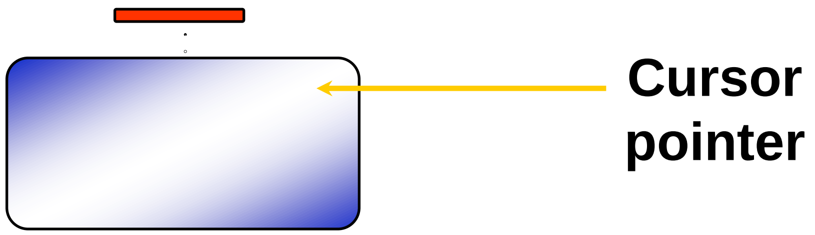
Continue until empty.

3. **Close the Cursor**

Controlling Explicit Cursors

1. Open the cursor
2. Fetch a row
3. Close the Cursor

3. Close the cursor.



Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
      select_statement;
```

Do not include the INTO clause in the cursor declaration.
If processing rows in a specific sequence is required, use the ORDER BY clause in the query.

Declaring the Cursor

Example:

```
DECLARE
  CURSOR emp_cursor IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR dept_cursor IS
    SELECT *
    FROM   departments
    WHERE  location_id = 170;
BEGIN
  ...
```

Opening the Cursor

Syntax:

```
OPEN cursor_name;
```

Open the cursor to execute the query and identify the active set.

If the query returns no rows, no exception is raised.

Use cursor attributes to test the outcome after a fetch.

Fetching Data from the Cursor

Syntax:

```
FETCH cursor_name INTO   [variable1, variable2, ...]
                                     | record_name];
```

Retrieve the current row values into variables.

Include the same number of variables.

Match each variable to correspond to the columns positionally.

Test to see whether the cursor contains rows.

Fetching Data from the Cursor

Example:

```
LOOP
  FETCH emp_cursor INTO v_empno,v_ename;
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  …
END LOOP;
```

Closing the Cursor

Syntax:

```
CLOSE            cursor_name;
```

Close the cursor after completing the processing of the rows.
Reopen the cursor, if required.
Do not attempt to fetch data from a cursor after it has been closed.

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far

The %ISOPEN Attribute

Fetch rows only when the cursor is open.
Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.
Example:

```
IF NOT emp_cursor%ISOPEN THEN
   OPEN emp_cursor;
END IF;
LOOP
  FETCH emp_cursor...
```

Controlling Multiple Fetches

Process several rows from an explicit cursor using a loop.

Fetch a row with each iteration.

Use explicit cursor attributes to test the success of each fetch.

The %NOTFOUND and %ROWCOUNT Attributes

Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows.

Use the %NOTFOUND cursor attribute to determine when to exit the loop.

Example

```
DECLARE
      v_empno employees.employee_id%TYPE;
      v_ename employees.last_name%TYPE;
      CURSOR emp_cursor IS
        SELECT employee_id, last_name
        FROM   employees;
    BEGIN
      OPEN emp_cursor;
      LOOP
        FETCH emp_cursor INTO v_empno, v_ename;
        EXIT WHEN emp_cursor%ROWCOUNT > 10 OR
                                  emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)
                                      ||' '|| v_ename);
      END LOOP;
      CLOSE emp_cursor;
END ;
```

Cursors and Records

Process the rows of the active set by fetching values into a PL/SQL RECORD.

```
DECLARE
  CURSOR emp_cursor IS
    SELECT     employee_id, last_name
    FROM       employees;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;….
```

emp_record

employee_id	last_name
100	King

Cursor FOR Loops

Syntax:

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

The cursor FOR loop is a shortcut to process explicit cursors.

Implicit open, fetch, exit, and close occur.

The record is implicitly declared.

SQL> SELECT * FROM temp ORDER BY col1; NUM_COL1 NUM_COL2 CHAR_COL -------- -------- --------- 1 100 i is odd 2 200 i is even 3 300 i is odd 4 400 i is e

Program to Print Odd even number

Sample Code

```
DECLARE
x NUMBER := 100;
BEGIN
FOR i IN 1..10 LOOP
IF MOD(i,2) = 0 THEN -- i is even
        INSERT INTO temp VALUES (i, x, 'i is even');
  ELSE
INSERT INTO temp VALUES (i, x, 'i is odd');
END IF;
x := x + 100;
END LOOP;
COMMIT;
END;
```

```
SQL> SELECT * FROM temp ORDER BY col1;
NUM_COL1 NUM_COL2 CHAR_COL
-------- -------- ---------
1 00   i is odd
2 200 i is even
3 300 i is odd
4 400 i is even
5 500 i is odd
6 600 i is even
7 700 i is odd
8 800 i is even
9 900 i is odd
10 1000 i is even
```

Cursor FOR Loops

```
Print a list of the employees who work for the sales
  department.
```

```
DECLARE
   CURSOR emp_cursor IS
      SELECT last_name, department_id
      FROM   employees;
BEGIN
   FOR emp_record IN emp_cursor LOOP
            -- implicit open and implicit fetch occur
      IF emp_record.department_id = 80 THEN
         ...
   END LOOP; -- implicit close occurs
END;
/
```

Cursor FOR Loops Using Subqueries

```
No need to declare the cursor.
Example:
```

```
BEGIN
  FOR emp_record IN (SELECT last_name, department_id
                     FROM   employees) LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.department_id = 80 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

BULK COLLECT

The BULK COLLECT clause lets you bulk-bind entire columns of Oracle data. That way, you can fetch all rows from the result set at once.

```
DECLARE
TYPE NumTab IS TABLE OF emp.empno%TYPE;
TYPE NameTab IS TABLE OF emp.ename%TYPE;
nums NumTab; names NameTab;
CURSOR c1 IS SELECT empno, ename FROM emp
WHERE job = 'CLERK';
BEGIN
OPEN c1;
FETCH c1 BULK COLLECT INTO nums, names;
... CLOSE c1;
END;
```

Example- ROWCOUNT & Top paid

Program To fetch the names and salaries of the five highest-paid employees:

The following PL/SQL block uses %ROWCOUNT to fetch the names and salaries of the five highest-paid employees:

```
DECLARE
CURSOR c1 is SELECT ename, empno, sal FROM emp ORDER BY sal
DESC; -- start with highest-paid employee
my_ename CHAR(10);
my_empno NUMBER(4);
my_sal NUMBER(7,2);
BEGIN
OPEN c1;
LOOP FETCH c1 INTO my_ename, my_empno, my_sal;
EXIT WHEN (c1%ROWCOUNT > 5) OR (c1%NOTFOUND); INSERT INTO temp
VALUES (my_sal, my_empno, my_ename);
COMMIT;
END LOOP;
CLOSE c1;
END;
```

Passing Parameters to a Cursor FOR Loop

You can pass parameters to the cursor in a cursor FOR loop.

In the following example, you pass a department number.

Then, you compute the total wages paid to employees in that department etc.

```
DECLARE
CURSOR emp_cursor(dnum NUMBER) IS SELECT sal, comm FROM emp WHERE deptno =
dnum;
total_wages NUMBER(11,2) := 0;
high_paid NUMBER(4) := 0;
higher_comm NUMBER(4) := 0;
BEGIN /* no of iterations = the no. of rows returned by emp_cursor. */
      FOR emp_record IN emp_cursor(20) LOOP
        emp_record.comm := NVL(emp_record.comm, 0);
        total_wages := total_wages + emp_record.sal + emp_record.comm;
                  IF emp_record.sal > 2000.00 THEN
                         high_paid := high_paid + 1;
                  END IF;
                 IF emp_record.comm > emp_record.sal THEN
                         higher_comm := higher_comm + 1;
                  END IF;
           END LOOP;
INSERT INTO temp VALUES (high_paid, higher_comm, 'Total Wages: ' ||
TO_CHAR(total_wages));
COMMIT;
END;
```

Managing Subprograms

Functions and Procedures

- Up until now, our code was in an anonymous block
- It was run immediately
- It is useful to put code in a function or procedure so it can be called several times
- Once we create a procedure or function in a Database, it will remain until deleted (like a table).

Creating Procedures

Modes:

- **IN:** procedure must be called with a value for the parameter. Value cannot be changed
- **OUT:** procedure must be called with a variable for the parameter. Changes to the parameter are seen by the user (i.e., call by reference)
- **IN OUT**: value can be sent, and changes to the parameter are seen by the user

Default Mode is: **IN**

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
IS|AS
PL/SQL Block;
```

Example- what does this do?

Table mylog

who	logon_ num
Pete	3
John	4
Joe	2

```
create or replace procedure
num_logged
(person IN mylog.who%TYPE,
 num OUT mylog.logon_num%TYPE)
IS
BEGIN
    select logon_num
    into num
    from mylog
    where who = person;
END;
/
```

Calling the Procedure

```
declare
    howmany  mylog.logon_num%TYPE;
begin
    num_logged(‘John',howmany);
    dbms_output.put_line(howmany);
end;
/
```

Creating a Function

Almost exactly like creating a procedure, but you supply a return type

```
CREATE [OR REPLACE] FUNCTION
function_name
 [(parameter1 [mode1] datatype1,
   parameter2 [mode2] datatype2,
   . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

A Function

```
create or replace function
rating_message(rating IN NUMBER)
return VARCHAR2
AS
BEGIN
  IF rating > 7 THEN
    return 'You are great';
  ELSIF rating >= 5 THEN
    return  'Not bad';
  ELSE
    return 'Pretty bad';
  END IF;
END;
/
```

NOTE THAT YOU DON'T SPECIFY THE SIZE

Calling the function

```
declare
    youRate:=9;
Begin
dbms_output.put_line(ratingMessage(youRate));
end;
/
```

Creating a function:

```
create or replace function squareFunc(num in number)
return number
is
BEGIN
return num*num;
End;
/
```

Using the function:

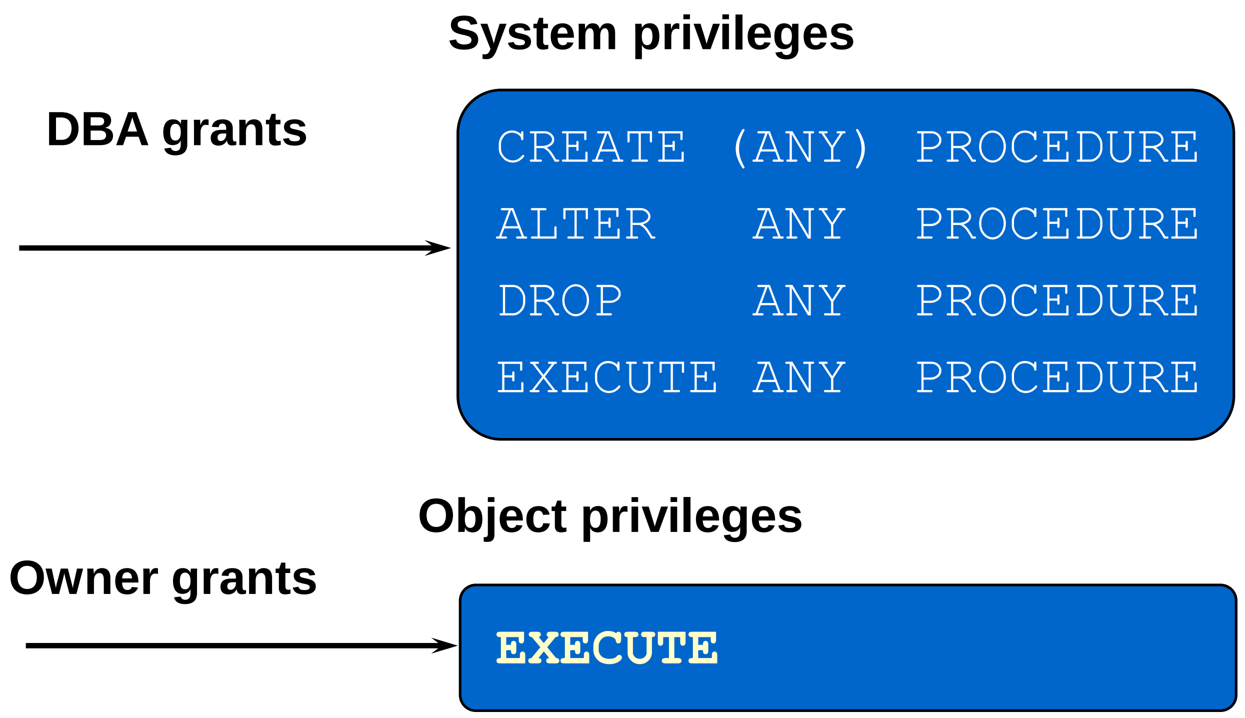
```
BEGIN
dbms_output.put_line(squareFunc(3.5));
END;
/
```

Errors in a Procedure

- When creating the procedure, if there are errors in its definition, they will not be shown
- To see the errors of a procedure called *myProcedure*, type
  - SHOW ERRORS PROCEDURE *myProcedure*

  in the SQLPLUS prompt
- For functions, type
  - SHOW ERRORS FUNCTION *myFunction*

Required Privileges



To be able to refer and access objects from a different schema in a subprogram, you must be granted access to the referred objects explicitly, not through a role.

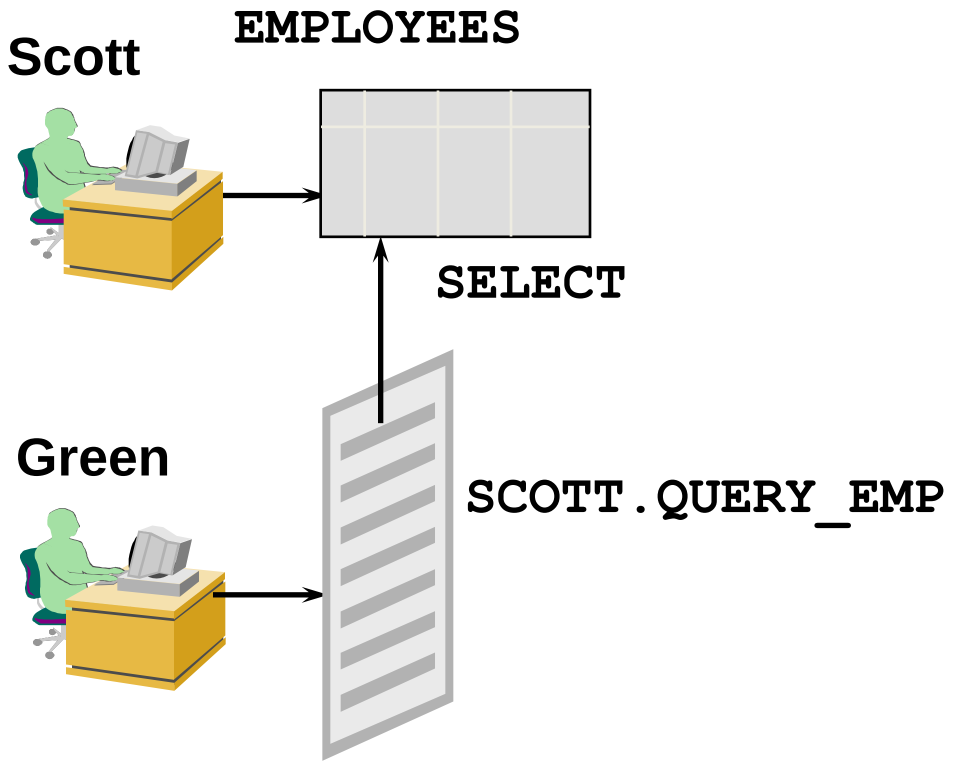
Granting Access to Data

Direct access:

```
GRANT SELECT
 ON   employees
 TO   scott;
Grant Succeeded.
```

Indirect access:

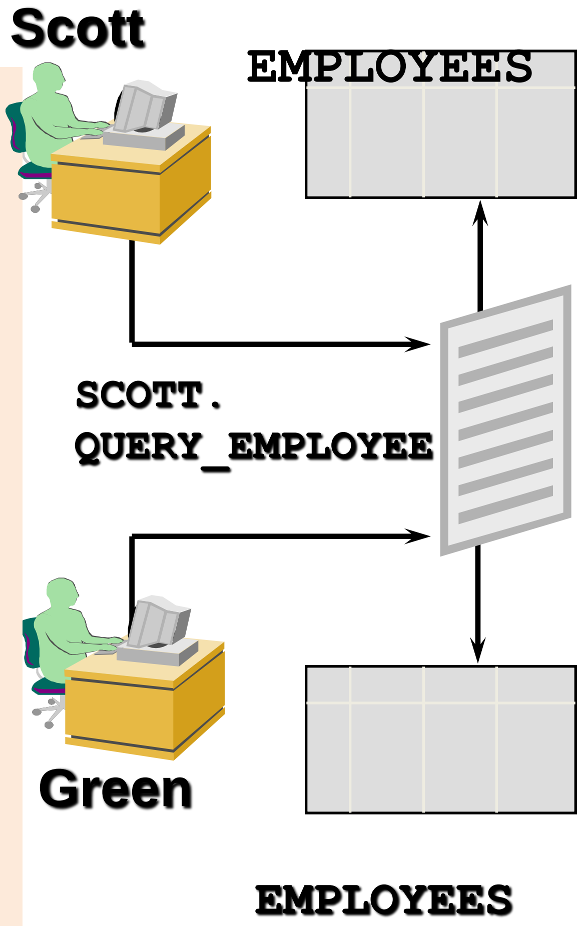
```
GRANT EXECUTE
 ON   query_emp
 TO   green;
Grant Succeeded.
```



The procedure executes with the privileges of the owner (default).

Using Invoker's-Rights

```
CREATE PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE,
 p_name OUT employees.last_name%TYPE,
 p_salary OUT employees.salary%TYPE,
 p_comm OUT
employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
  SELECT last_name, salary,
         commission_pct
   INTO  p_name, p_salary, p_comm
   FROM  employees
   WHERE employee_id=p_id;
END query_employee;
/
```



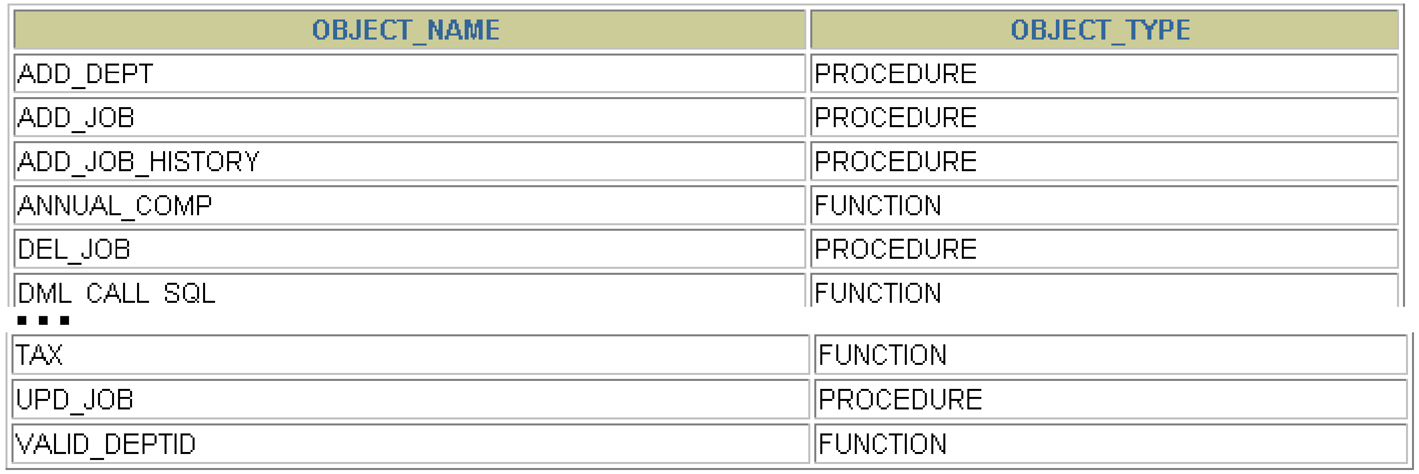
The procedure executes with the privileges of the user.

USER_OBJECTS

Column	Column Description
OBJECT_NAME	Name of the object
OBJECT_ID	Internal identifier for the object
OBJECT_TYPE	Type of object, for example, TABLE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
CREATED	Date when the object was created
LAST_DDL_TIME	Date when the object was last modified
TIMESTAMP	Date and time when the object was last recompiled
STATUS	VALID or INVALID

List All Procedures and Functions

```
SELECT object_name, object_type
FROM   user_objects
WHERE  object_type in ('PROCEDURE','FUNCTION')
ORDER  BY object_name;
```



OBJECT_NAME	OBJECT_TYPE
ADD_DEPT	PROCEDURE
ADD_JOB	PROCEDURE
ADD_JOB_HISTORY	PROCEDURE
ANNUAL_COMP	FUNCTION
DEL_JOB	PROCEDURE
DML_CALL_SQL	FUNCTION
…	
TAX	FUNCTION
UPD_JOB	PROCEDURE
VALID_DEPTID	FUNCTION

24 rows selected.

USER_SOURCE Data Dictionary View

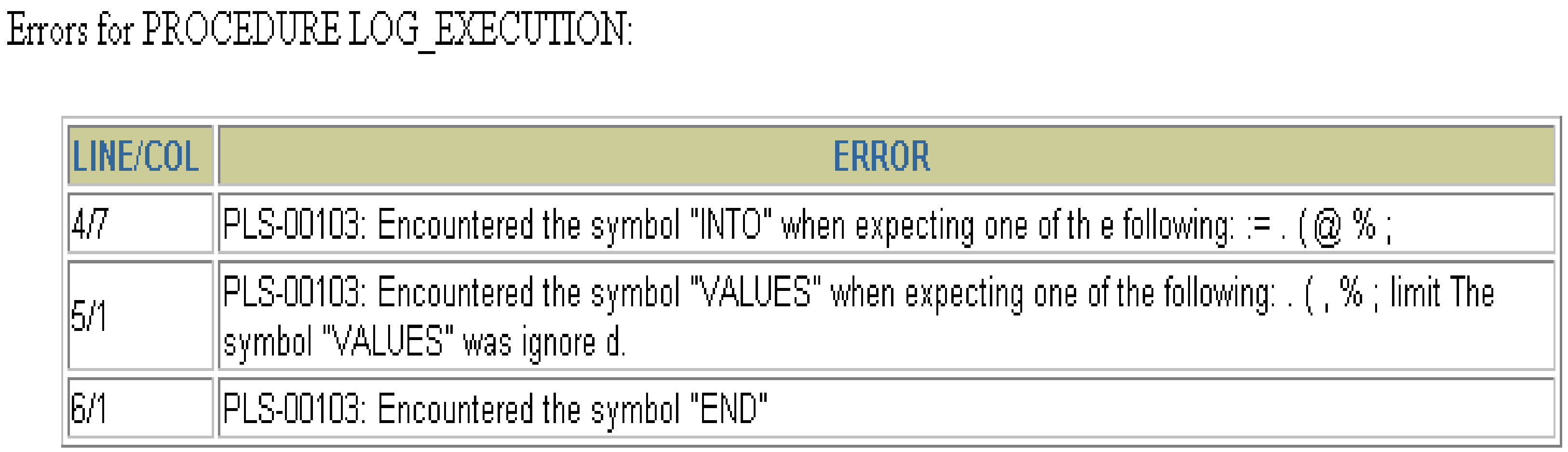
Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY
LINE	Line number of the source code
TEXT	Text of the source code line

USER_ERRORS

Column	Column Description
`NAME`	Name of the object
`TYPE`	Type of object, for example, `PROCEDURE`, `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `TRIGGER`
`SEQUENCE`	Sequence number, for ordering
`LINE`	Line number of the source code at which the error occurs
`POSITION`	Position in the line at which the error occurs
`TEXT`	Text of the error message

List Compilation Errors by Using SHOW ERRORS

```
SHOW ERRORS PROCEDURE log_execution
```



Errors for PROCEDURE LOG_EXECUTION:

LINE/COL	ERROR
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of th e following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignore d.
6/1	PLS-00103: Encountered the symbol "END"

Creating Packages

Objectives

After completing this Section, you should be able to do the following:

- Describe packages and list their possible components
- Create a package to group together related variables, cursors, constants, exceptions, procedures, and functions
- Designate a package construct as either public or private
- Invoke a package construct
- Describe a use for a bodiless package

Overview of Packages

Packages:

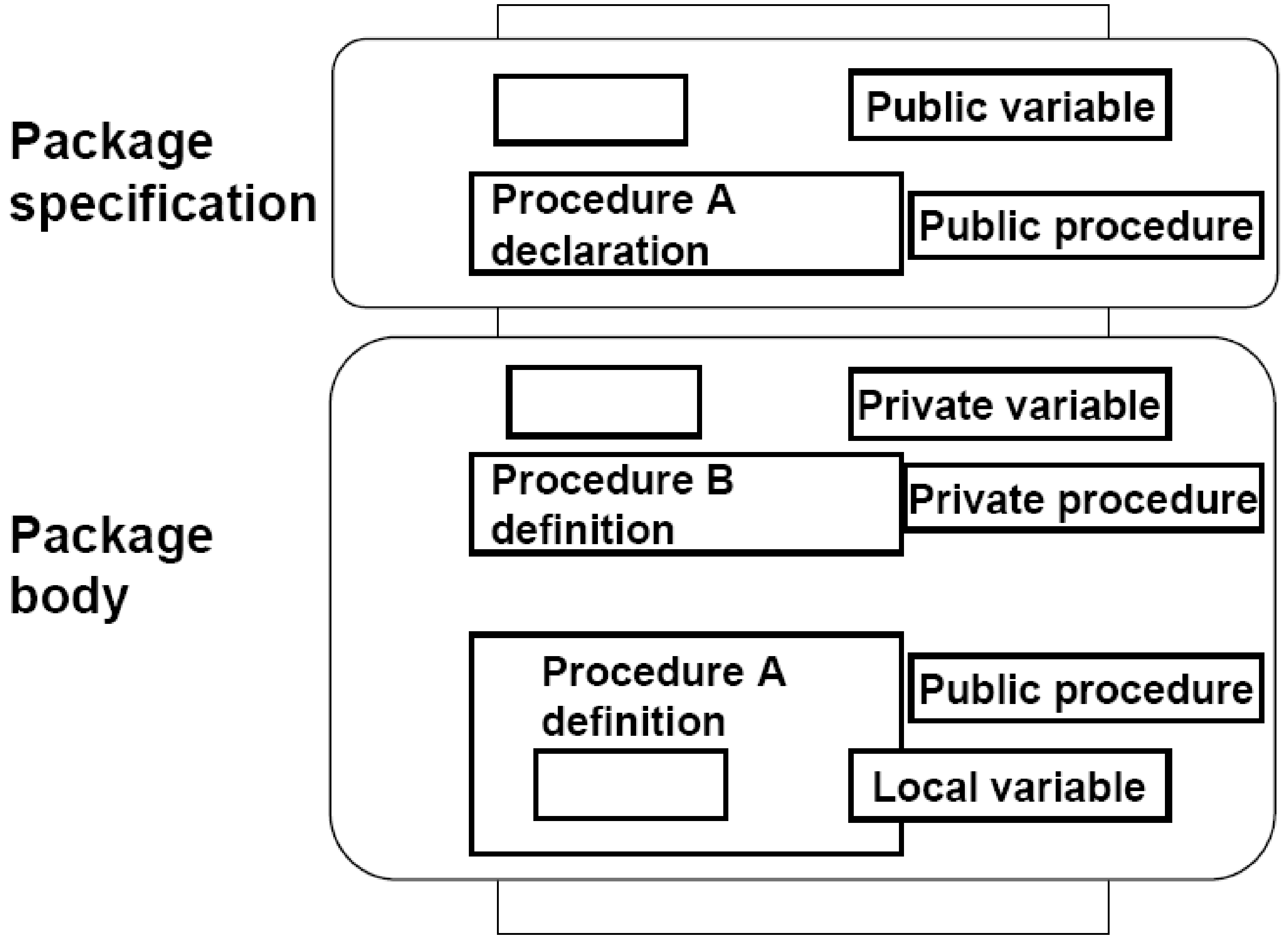
- **Group logically related PL/SQL types, items, and subprograms**
- **Consist of two parts:**

– Specification

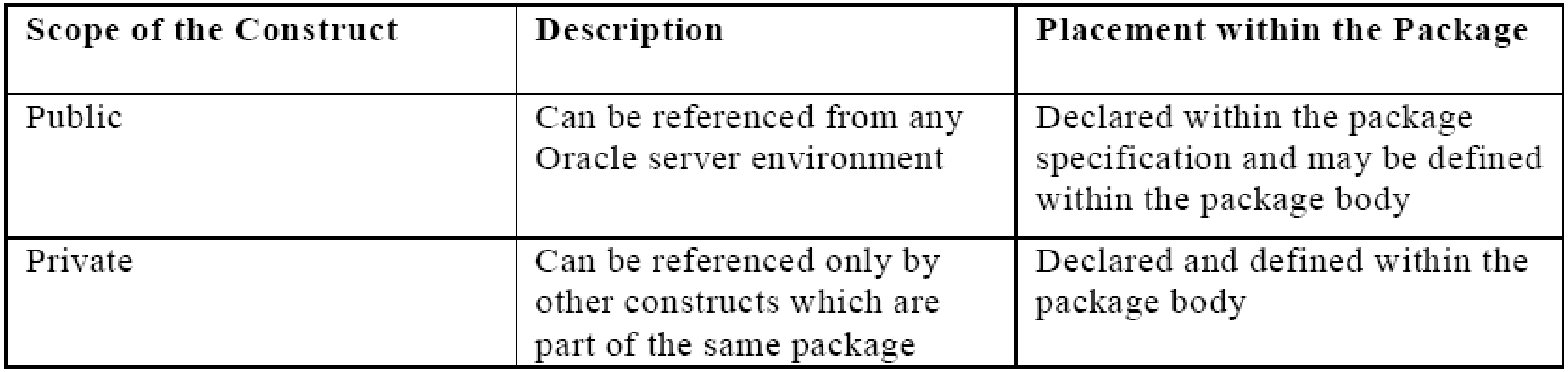
– Body

- **Cannot be invoked, parameterized, or nested**
- **Allow the Oracle server to read multiple objects into memory at once**

Components of a Package

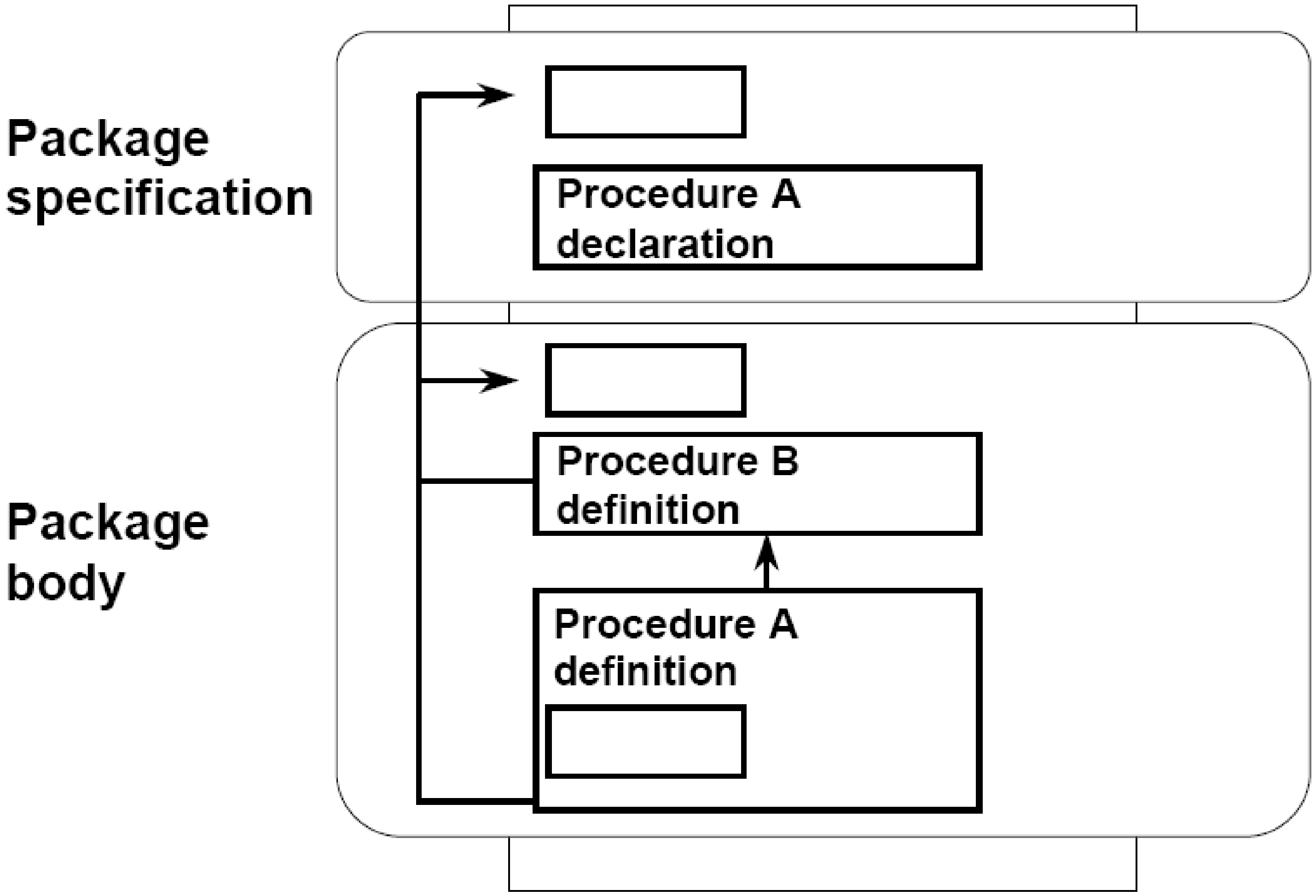


Scope



Scope of the Construct	Description	Placement within the Package
Public	Can be referenced from any Oracle server environment	Declared within the package specification and may be defined within the package body
Private	Can be referenced only by other constructs which are part of the same package	Declared and defined within the package body

Referencing Package Objects



Developing a Package

Developing a Package

- Saving the text of the CREATE PACKAGE statement in two different SQL files facilitates later modifications to the package.

- A package specification can exist without a package body, but a package body cannot exist without a package specification.

How to Develop a Package

There are three basic steps to developing a package, similar to those steps that are used to develop a stand-alone procedure.

1. Write the text of the CREATE PACKAGE statement within a SQL script file to create the package specification and run the script file. The source code is compiled into P code and is stored within the data dictionary.

2. Write the text of the CREATE PACKAGE BODY statement within a SQL script file to create the package body and run the script file. The source code is compiled into P code and is also stored within the data dictionary.

3. Invoke any public construct within the package from an Oracle server environment.

Creating the Package Specification

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
IS|AS
    public type and item declarations
    subprogram specifications
END package_name;
```

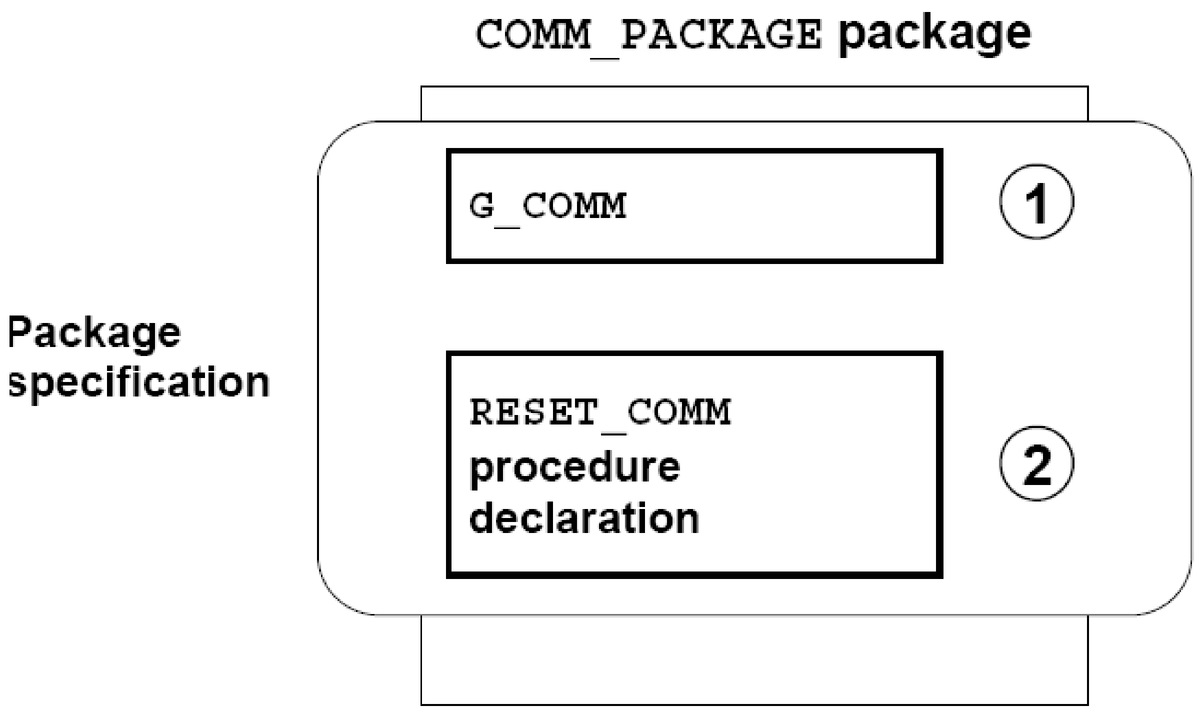
- **The REPLACE option drops and recreates the package specification.**
- **Variables declared in the package specification are initialized to NULL by default.**
- **All the constructs declared in a package specification are visible to users who are granted privileges on the package.**

Syntax Definition

```
CREATE [OR REPLACE] PACKAGE package_name
IS|AS
    public type and item declarations
    subprogram specifications
END package_name;
```

Parameter	Description
package_name	Name the package
public type and item declarations	Declare variables, constants, cursors, exceptions, or types
subprogram specifications	Declare the PL/SQL subprograms

Declaring Public Constructs



Creating a Package Specification: Example

```
CREATE OR REPLACE PACKAGE comm_package IS
  g_comm NUMBER := 0.10;  --initialized to 0.10
  PROCEDURE reset_comm
  (p_comm   IN  NUMBER);
END comm_package;
/
```

```
Package created.
```

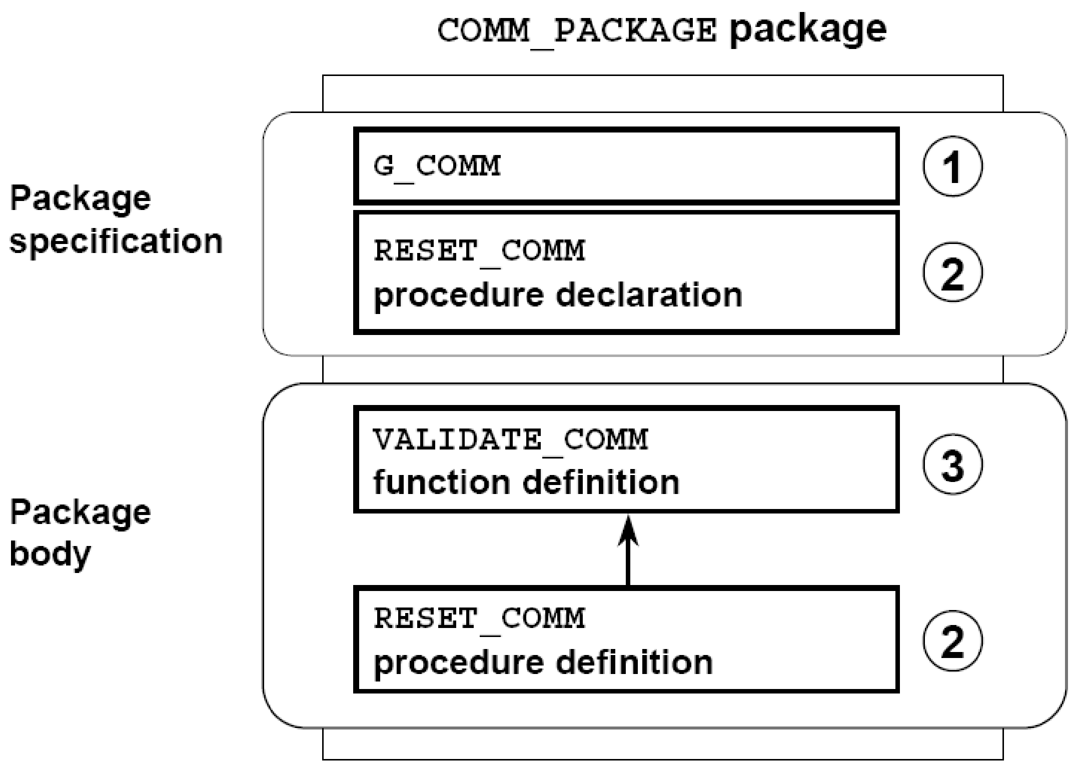
- G_COMM is a global variable and is initialized to 0.10.
- RESET_COMM is a public procedure that is implemented in the package body.

Creating the Package Body

```
CREATE [OR REPLACE] PACKAGE BODY package_name
IS|AS
    private type and item declarations
    subprogram bodies
END package_name;
```

- The REPLACE option drops and recreates the package body.
- Identifiers defined only in the package body are private constructs. These are not visible outside the package body.
- All private constructs must be declared before they are used in the public constructs.

Public and Private Constructs


COMM_PACKAGE package
Package specification
Package body
G_COMM
1
RESET_COMM procedure declaration
2
VALIDATE_COMM function definition
3
RESET_COMM procedure definition
2

Creating a Package Body: Example

comm_pack.sql

```
CREATE OR REPLACE PACKAGE BODY comm_package
IS
   FUNCTION  validate_comm (p_comm IN NUMBER)
    RETURN BOOLEAN
   IS
     v_max_comm    NUMBER;
   BEGIN
     SELECT    MAX(commission_pct)
      INTO     v_max_comm
      FROM     employees;
     IF   p_comm > v_max_comm THEN RETURN(FALSE);
     ELSE   RETURN(TRUE);
     END IF;
   END validate_comm;
...
```

Creating a Package Body: Example

comm_pack.sql

```
  PROCEDURE  reset_comm (p_comm   IN  NUMBER)
  IS
  BEGIN
   IF  validate_comm(p_comm)
    THEN    g_comm:=p_comm;  --reset global variable
   ELSE
    RAISE_APPLICATION_ERROR(-20210,'Invalid commission');
   END IF;
  END reset_comm;
END comm_package;
/
```

```
Package body created.
```

Invoking Package Constructs

Example 1: Invoke a function from a procedure within the same package.

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
   . . .
 PROCEDURE reset_comm
  (p_comm   IN  NUMBER)
 IS
 BEGIN
  IF validate_comm(p_comm)
  THEN g_comm := p_comm;
  ELSE
    RAISE_APPLICATION_ERROR
         (-20210, 'Invalid commission');
  END IF;
 END reset_comm;
END comm_package;
```

Invoking Package Constructs

Example 2: Invoke a package procedure from *i*SQL*Plus.

```
EXECUTE comm_package.reset_comm(0.15)
```

Example 3: Invoke a package procedure in a different schema.

```
EXECUTE scott.comm_package.reset_comm(0.15)
```

Example 4: Invoke a package procedure in a remote database.

```
EXECUTE comm_package.reset_comm@ny(0.15)
```

Declaring a Bodiless Package

```
CREATE OR REPLACE PACKAGE global_consts IS
  mile_2_kilo     CONSTANT  NUMBER  :=  1.6093;
  kilo_2_mile     CONSTANT  NUMBER  :=  0.6214;
  yard_2_meter    CONSTANT  NUMBER  :=  0.9144;
  meter_2_yard    CONSTANT  NUMBER  :=  1.0936;
END global_consts;
/

EXECUTE DBMS_OUTPUT.PUT_LINE('20 miles = '||20*
    global_consts.mile_2_kilo||' km')
```

```
Package created.
20 miles = 32.186 km
PL/SQL procedure successfully completed.
```

Referencing a Public Variable froma Stand-Alone Procedure

```
CREATE OR REPLACE PROCEDURE meter_to_yard
            (p_meter IN NUMBER, p_yard OUT NUMBER)
IS
BEGIN
  p_yard := p_meter * global_consts.meter_2_yard;
END meter_to_yard;
/
VARIABLE yard NUMBER
EXECUTE  meter_to_yard (1, :yard)
PRINT yard
```

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

```
                 YARD
                                1.0936
```

Removing Packages

To remove the package specification and the body, use the following syntax:

DROP PACKAGE *package_name;*

To remove the package body, use the following syntax:

DROP PACKAGE BODY *package_name;*

Guidelines for Developing Packages

- Construct packages for general use.
- Define the package specification before the body.
- The package specification should contain only those constructs that you want to be public.
- Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.
- Changes to the package specification require recompilation of each referencing subprogram.
- The package specification should contain as few constructs as possible.

Advantages of Packages

- Modularity: Encapsulate related constructs.
- Easier application design: Code and compile specification and body separately.
- Hiding information:
  - Only the declarations in the package specification are visible and accessible to applications.
  - Private constructs in the package body are hidden and inaccessible.
  - All coding is hidden in the package body.

Advantages of Packages

- **Added functionality: Persistency of variables and cursors**

- **Better performance:**

– The entire package is loaded into memory when the package is first referenced.

– There is only one copy in memory for all users.

– The dependency hierarchy is simplified.

- **Overloading: Multiple subprograms of the same name**

Creating a One-Time-Only Procedure

```
CREATE OR REPLACE PACKAGE taxes
IS
    tax   NUMBER;
  ...  -- declare all public procedures/functions
END  taxes;
/
```

```
CREATE OR REPLACE PACKAGE BODY taxes
IS
  ... -- declare all private variables
  ... -- define public/private procedures/functions
BEGIN
   SELECT   rate_value
   INTO     tax
   FROM     tax_rates
   WHERE    rate_name = 'TAX';
END taxes;
/
```

Restrictions on Package Functions Used in SQL

A function called from:

- A query or DML statement can not end the current transaction, create or roll back to a savepoint, or ALTER the system or session.
- A query statement or a parallelized DML statement can not execute a DML statement or modify the database.
- A DML statement can not read or modify the particular table being modified by that DML statement.

Note: Calls to subprograms that break the above restrictions are not allowed

PL/SQL Tables and Records in Packages

```
CREATE OR REPLACE PACKAGE emp_package IS
  TYPE emp_table_type IS TABLE OF employees%ROWTYPE
     INDEX BY BINARY_INTEGER;
  PROCEDURE read_emp_table
               (p_emp_table OUT emp_table_type);
END emp_package;
/
```

```
CREATE OR REPLACE PACKAGE BODY emp_package IS
  PROCEDURE read_emp_table
               (p_emp_table OUT emp_table_type) IS
  i BINARY_INTEGER := 0;
  BEGIN
    FOR emp_record IN (SELECT * FROM employees)
    LOOP
      p_emp_table(i) := emp_record;
      i:= i+1;
    END LOOP;
  END read_emp_table;
END emp_package;
/
```

PL/SQL Tables and Records in Packages

```
DECLARE
  v_emp_table emp_package.emp_table_type;
  BEGIN
    emp_package.read_emp_table(v_emp_table);
    DBMS_OUTPUT.PUT_LINE('An example: '||v_emp_table(4).last_name);
END;
/
```

```
An example: Ernst
PL/SQL procedure successfully completed.
```

Summary

In this Section, you should have learned how to:

- Improve organization, management, security, and performance by using packages
- Group related procedures and functions together in a package
- Change a package body without affecting a package specification
- Grant security access to the entire package
- Hide the source code from users
- Load the entire package into memory on the first call
- Reduce disk access for subsequent calls
- Provide identifiers for the user session

Summary

Command	Task
`CREATE [OR REPLACE] PACKAGE`	Create (or modify) an existing package specification
`CREATE [OR REPLACE] PACKAGE BODY`	Create (or modify) an existing package body
`DROP PACKAGE`	Remove both the package specification and the package body
`DROP PACKAGE BODY`	Remove the package body only

Practices

1. Create a package specification and body called JOB_PACK. (You can save the package body and specification in two separate files.) This package contains your ADD_JOB, UPD_JOB, and DEL_JOB procedures, as well as your Q_JOB function.

Note: Use the code in your previously saved script files when creating the package.

a. Make all the constructs public.

Note: Consider whether you still need the stand-alone procedures and functions you just packaged.

b. Invoke your ADD_JOB procedure by passing values IT_SYSAN and SYSTEMS ANALYST as parameters.

c. Query the JOBS table to see the result.

2. Create and invoke a package that contains private and public constructs.

a. Create a package specification and package body called EMP_PACK that contains your NEW_EMP procedure as a public construct, and your VALID_DEPTID function as a private construct. (You can save the specification and body into separate files.)

b. Invoke the NEW_EMP procedure, using 15 as a department number. Because the department ID 15 does not exist in the DEPARTMENTS table, you should get an error message as specified in the exception handler of your procedure.

c. Invoke the NEW_EMP procedure, using an existing department ID 80.

3. **a**. Create a package called CHK_PACK that contains the procedures CHK_HIREDATE and CHK_DEPT_MGR. Make both constructs public. (You can save the specification and body into separate files.) The procedure CHK_HIREDATE checks whether an employee's hire date is within the following range: [SYSDATE - 50 years, SYSDATE + 3 months].

Note:

• If the date is invalid, you should raise an application error with an appropriate message indicating why the date value is not acceptable.

• Make sure the time component in the date value is ignored.

• Use a constant to refer to the 50 years boundary.

• A null value for the hire date should be treated as an invalid hire date. The procedure CHK_DEPT_MGR checks the department and manager combination for a given employee. The CHK_DEPT_MGR procedure accepts an employee ID and a manager ID. The procedure checks that the manager and employee work in the same department. The procedure also checks that the job title of the manager ID provided is MANAGER.

Note: If the department ID and manager combination is invalid, you should raise an application error with an appropriate message.

b. Test the CHK_HIREDATE procedure with the following command:

EXECUTE chk_pack.chk_hiredate('01-JAN-47')

What happens, and why?

c. Test the CHK_HIREDATE procedure with the following command:

EXECUTE chk_pack.chk_hiredate(NULL)

What happens, and why?

d. Test the CHK_DEPT_MGR procedure with the following command:

EXECUTE chk_pack.chk_dept_mgr(117,100)

What happens, and why?

Handling Exceptions

Objectives

After completing this Section, you should be able to do the following:

- Define PL/SQL exceptions
- Recognize unhandled exceptions
- List and use different types of PL/SQL exception handlers
- Trap unanticipated errors
- Describe the effect of exception propagation in nested blocks
- Customize PL/SQL exception messages

Handling Exceptions with PL/SQL

- An Exception is an identifier in PL/SQL that is raised during execution.
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

Exception Types

- Predefined Oracle Server
- Nonpredefined Oracle Server

Implicitly raised

- User-defined

Explicitly raised

Trapping Exceptions

Syntax:

```
EXCEPTION
    WHEN exception1 [OR exception2 . . .] THEN
        statement1;
        statement2;
    [WHEN exception3 [OR exception4 . . .] THEN
        statement1;
        statement2;
        . . .]
    [WHEN OTHERS THEN
        statement1;
        statement2;
        . . .]
```

Trapping Predefined Oracle Server Errors

- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

Predefined Exceptions

Exception Name	Oracle server Error Number	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails
LOGIN_DENIED	ORA-01017	Logging on to Oracle with an invalid username or password
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data
NOT_LOGGED_ON	ORA-01012	PL/SQL program issues a database call without being connected to Oracle
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem

Predefined Exceptions

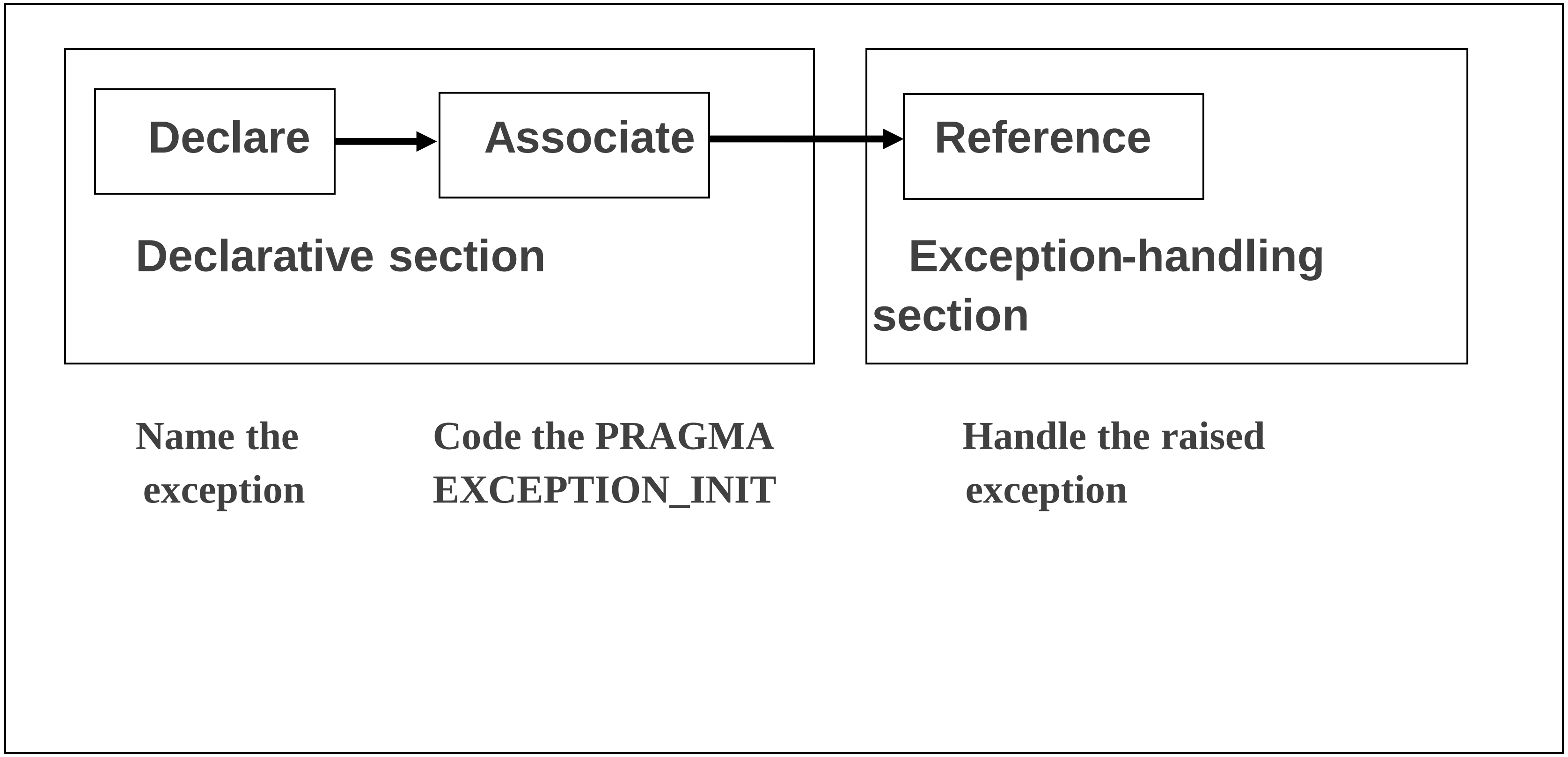
ROWTYPE_MISMATCH	ORA-06504	Host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types
STORAGE_ERROR	ORA-06500	PL/SQL ram out of memory is corrupted .
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or varray element using an index number that is outside the legal range (-1 for example)
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID fails because the character string does not represent a valid ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while Oracle is waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size- constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Predefined Exceptions

Syntax:

```
BEGIN

EXCEPTION
        WHEN NO_DATA_FOUND THEN
              statement1;
             statement2;
        WHEN TOO_MANY_ROWS THEN
              statement1;
        WHEN OTHERS THEN
              statement1;
              statement2;
             statement3;
END;
```

Trapping Nonpredefined Oracle Server Errors


Declare
Associate
Reference
Declarative section
Exception-handling section
Name the exception
Code the PRAGMA EXCEPTION_INIT
Handle the raised exception

Functions for Trapping Exceptions

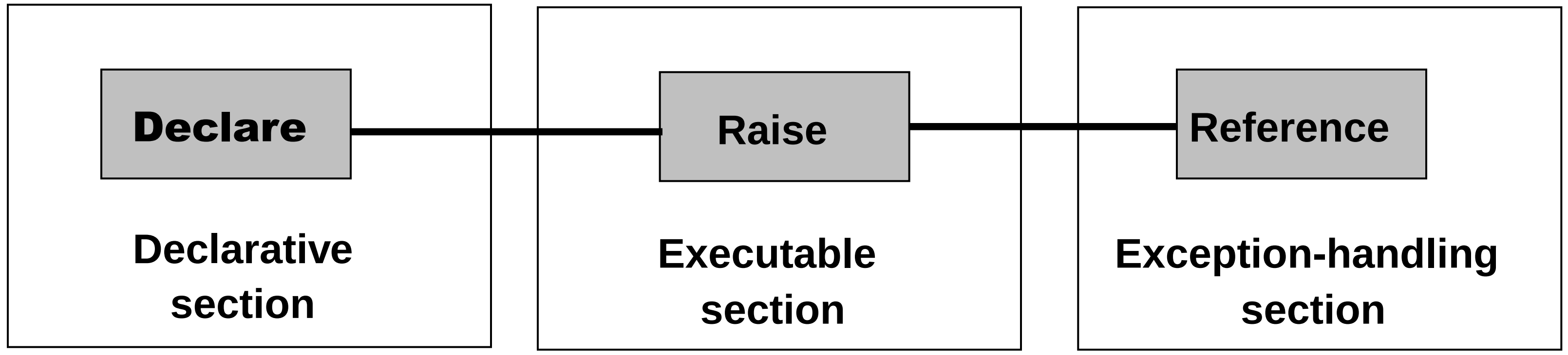
- SQLCODE: Returns the numeric value for the error code
- SQLERRM: Returns the message associated with the error number

Functions for Trapping Exceptions

Example:

```
  DECLARE
     v_error_code              NUMBER;
     v_error_message           VARCHAR2(255);
 BEGIN

 EXCEPTION
      WHEN OTHERS THEN
              ROLLBACK;
              v_error_code:=  SQLCODE ;
              v_error_message :=  SQLERRM ;
            INSERT INTO errors
            VALUES (v_error_code, v_error_message);
 END;
```

Trapping User-Defined Exceptions



Name the exception.

Explicitly raise the exception by using the RAISE statement.

Handle the raised exception.

User-Defined Exceptions

Example:

```
DEFINE p_department_desc = 'Information Technology '
DEFINE P_department_number = 300

    DECLARE
        e_invalid_department EXCEPTION;                               1
    BEGIN
        UPDATE    departments
        SET department_name = '&p_department_desc'
        WHERE     department_id = &p_department_number;
        IF SQL%NOTFOUND THEN
                  RAISE e_invalid_department;                         2
        END IF;
        COMMIT;
    EXCEPTION
        e_invalid_department                                          3
        DBMS_OUTPUT. PUT_LINE ('No such department id.');
    END;
```

The RAISE_APPLICATION_ERROR

Procedure

Syntax:

```
raise_application_error (error_number,
                         message[, {TRUE | FALSE}]);
```

- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

Practices

1. Write a PL/SQL block to select the name of the employee with a given salary value.

a. Use the DEFINE command to provide the salary. Pass the value to the PL/SQL block through a iSQL*Plus substitution variable. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the MESSAGES table the message “More than one employee with a salary of <salary>.”

b. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the MESSAGES table the message “No employee with a salary of <salary>.”

c. If the salary entered returns only one row, insert into the MESSAGES table the employee’s name and the salary amount.

d. Handle any other exception with an appropriate exception handler and insert into the MESSAGES table the message “Some other error occurred.”

Oracle Supplied Packages

Extend the functionality of the database

Some examples of packages:

- DBMS_JOB: for task scheduling
- DBMS_PIPE: for communication between sessions
- DBMS_OUTPUT: display messages to the session output device
- UTL_HTTP: makes HTTP callouts.
- Many others…

Triggers

Stored procedure

Executed automatically when:

- data modification (DML Trigger)
  - INSERT, UPDATE, UPDATE column or DELETE
- schema modification (DDL Trigger)
- system event, user logon/logoff (System Trigger)

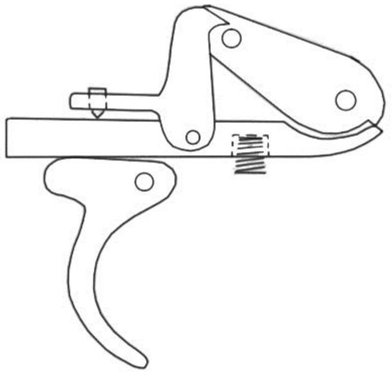
Basic DML triggers types:

- BEFORE statement
- BEFORE each row modification
- AFTER each row modification
- AFTER statement
- INSTEAD OF - to enable data modification by views

When To Use Triggers

- Automatic data generation
  - Auditing (logging), statistics
  - Derived data
  - Data replication
- Special referential constrains
  - Complex logic
  - Distributed constrains
  - Time based constrains
- Updates of complex views
- Triggers may introduce hard to spot interdependencies to a database schema

Trigger Body



Built like a PL/SQL procedure

Additionally:

Type of the triggering event can be determined inside the trigger using conditional predicators

```
IF inserting THEN … END IF;
```

Old and new row values are accessible via `:old` and `:new` qualifiers (record type variables)

Trigger Example

```
CREATE OR REPLACE TRIGGER audit_sal
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
   INSERT INTO emp_audit
   VALUES( :old.employee_id, SYSDATE, :new.salary, :old.salary );
   COMMIT;
END;
/
```

Jobs

Schedule + PL/SQL subprogram

Many scheduling modes

Creation

- Using **DBMS_SCHEDULER** internal package
  - Alternative DBMS_JOB is old and should by avoided
- Privileges needed
  - **execute** on DBMS_SCHEDULER
  - **create job**

Jobs example

Daily execution (everyday at 12) of *my_saved_procedure*

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
   job_name          =>  'my_new_job1',
   program_name      =>  'my_saved_procedure',
   repeat_interval   =>  'FREQ=DAILY;BYHOUR=12',
   comments          =>  'Daily at noon');
END;
/
```

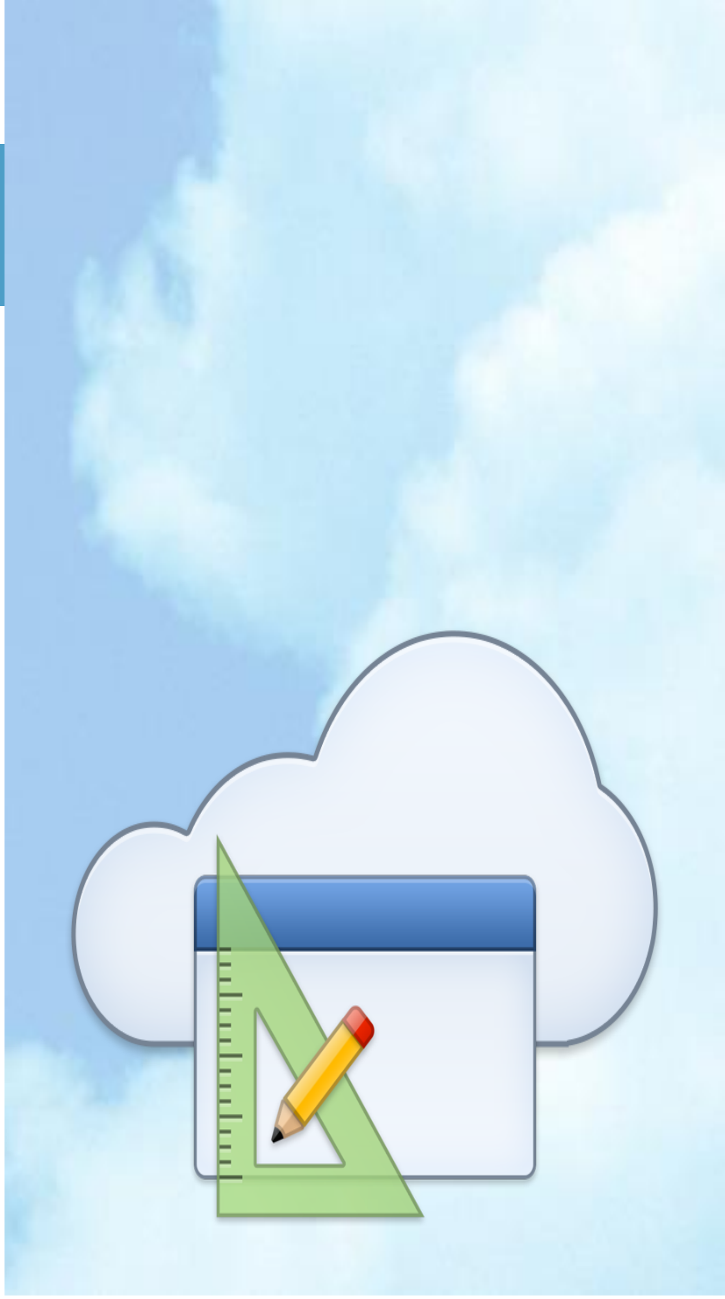
Advantages of PL/SQL

- Tightly integrated with SQL
- Reduced network traffic
- Portability - easy deployment and distribution
- Data layer separated from client language
  - Modification without changing of application code
  - Can be shared by many platform
- Server-side periodical data maintenance (jobs)

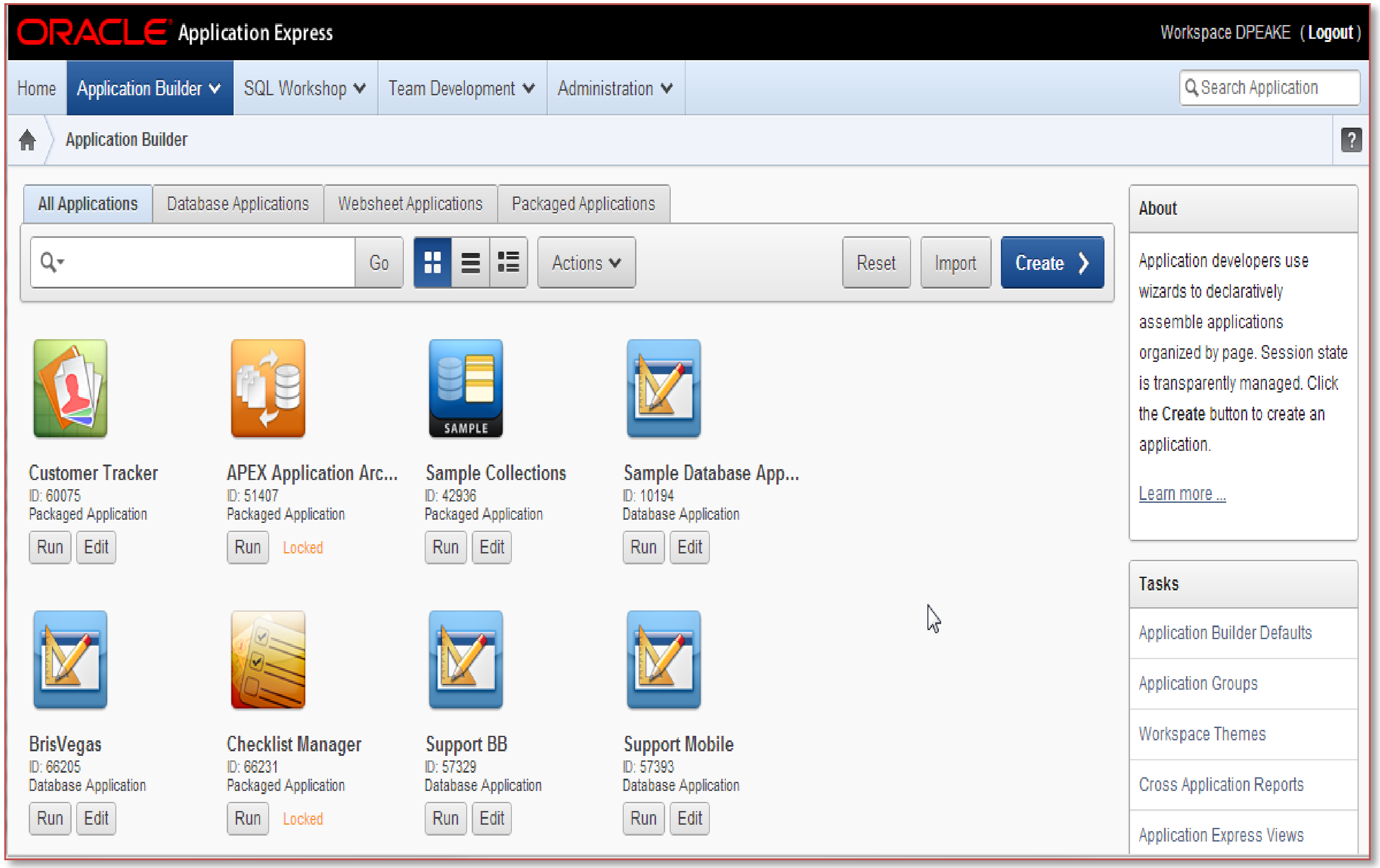
Questions?

Thank you!

Oracle Application Express

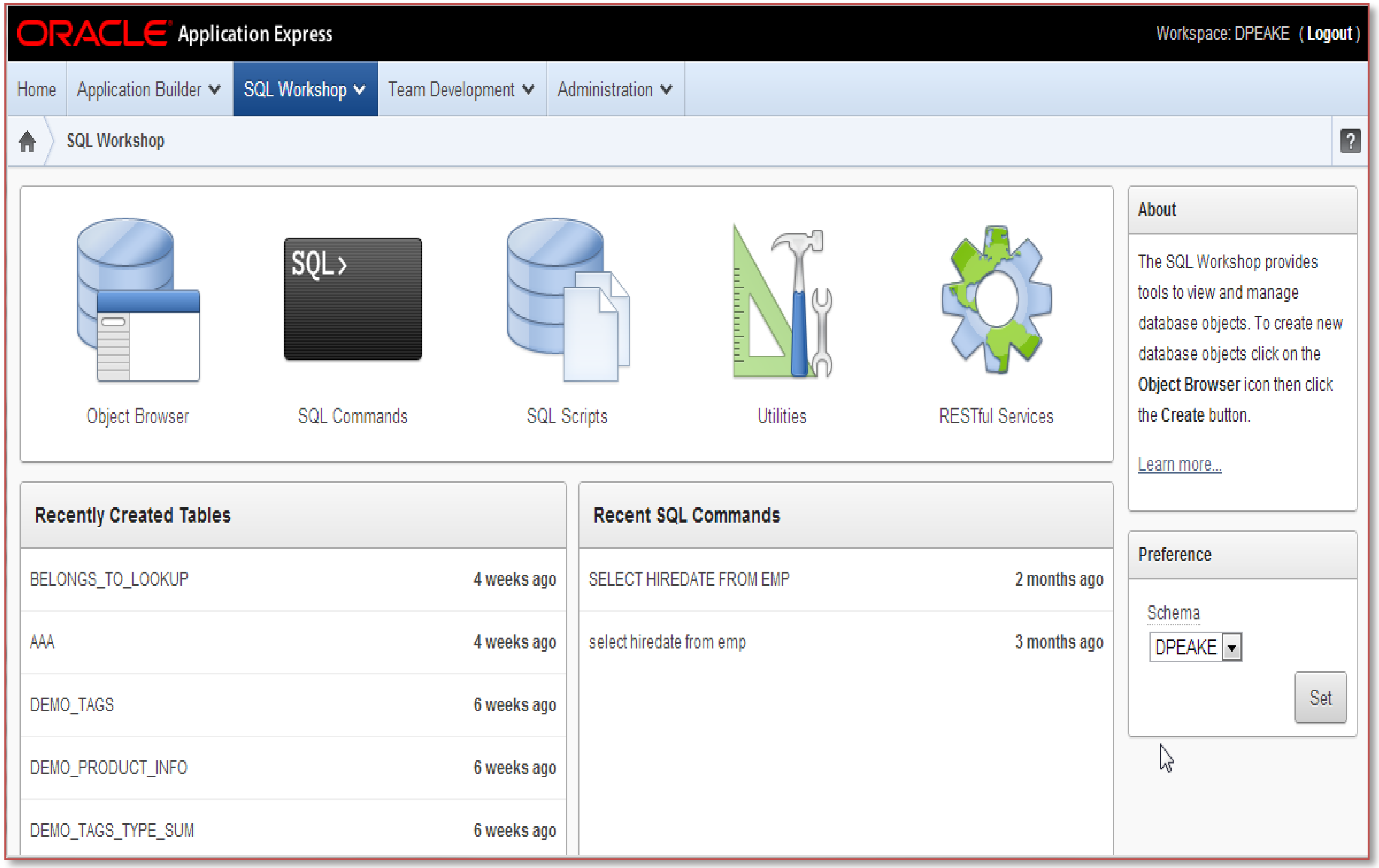


Oracle Application Express - Components



Application developers use wizards to declaratively assemble applications organized by page. Session state is transparently managed. Click the **Create** button to create an application.

Oracle Application Express - Components



About

The SQL Workshop provides tools to view and manage database objects. To create new database objects click on the **Object Browser** icon then click the **Create** button.

Learn more...

Recently Created Tables

BELONGS_TO_LOOKUP	4 weeks ago
AAA	4 weeks ago
DEMO_TAGS	6 weeks ago
DEMO_PRODUCT_INFO	6 weeks ago
DEMO_TAGS_TYPE_SUM	6 weeks ago

Recent SQL Commands

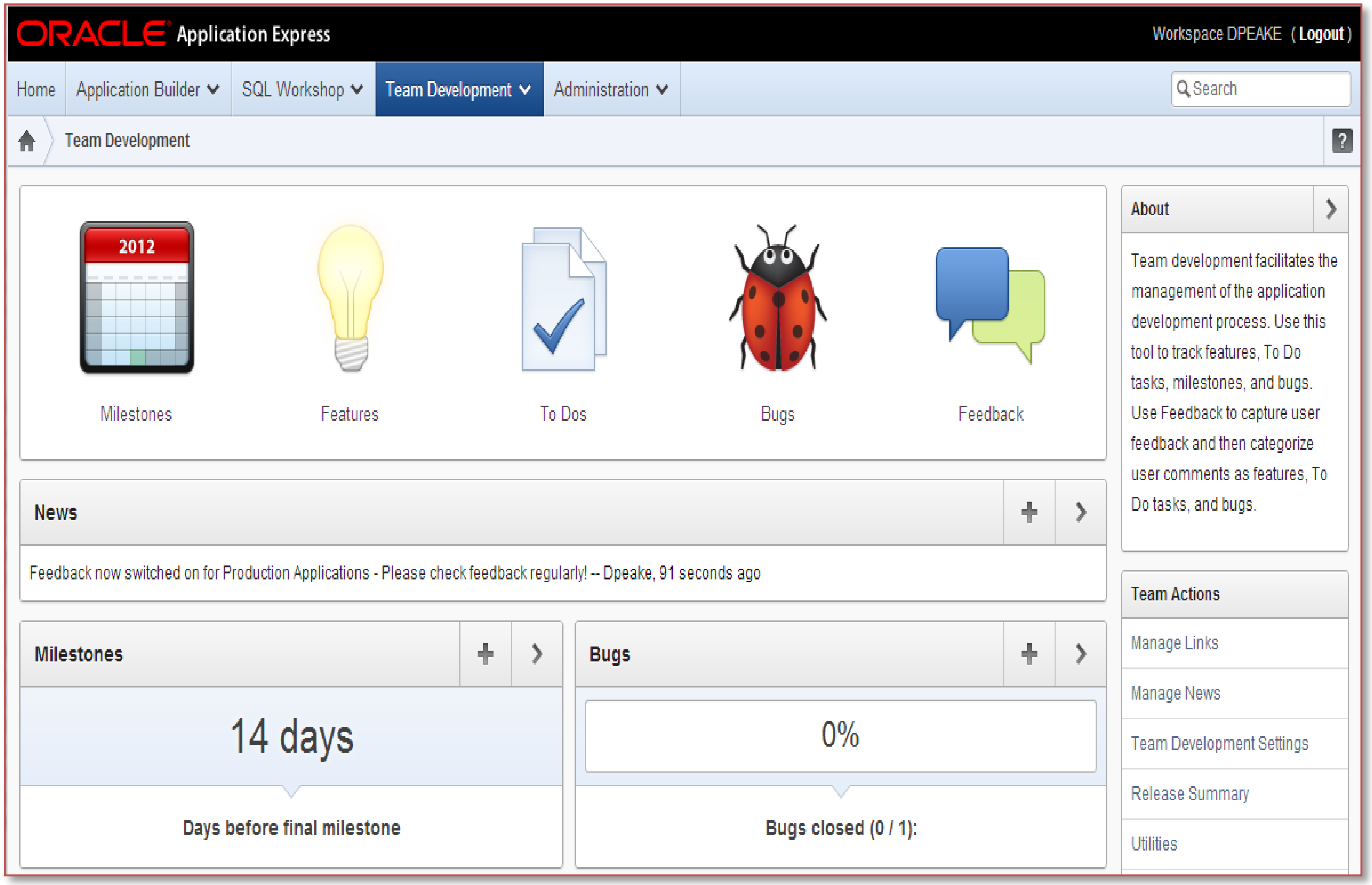
SELECT HIREDATE FROM EMP	2 months ago
select hiredate from emp	3 months ago

Preference

Schema: DPEAKE

Set

Oracle Application Express - Components



Team development facilitates the management of the application development process. Use this tool to track features, To Do tasks, milestones, and bugs. Use Feedback to capture user feedback and then categorize user comments as features, To Do tasks, and bugs.

News

Feedback now switched on for Production Applications - Please check feedback regularly! -- Dpeake, 91 seconds ago

Milestones

14 days

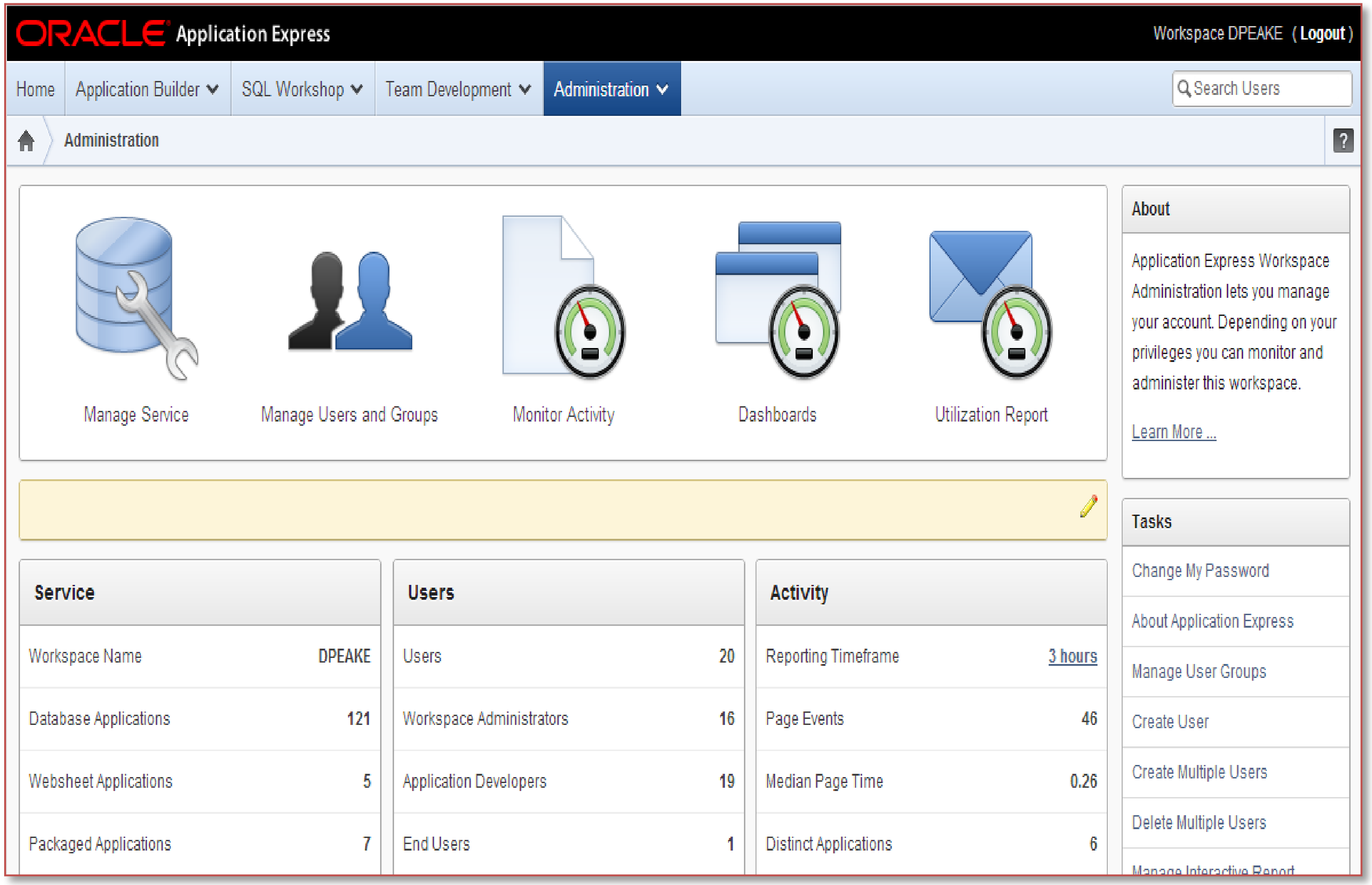
Days before final milestone

Bugs

0%

Bugs closed (0 / 1):

Oracle Application Express - Components



About

Application Express Workspace Administration lets you manage your account. Depending on your privileges you can monitor and administer this workspace.

Learn More ...

Tasks

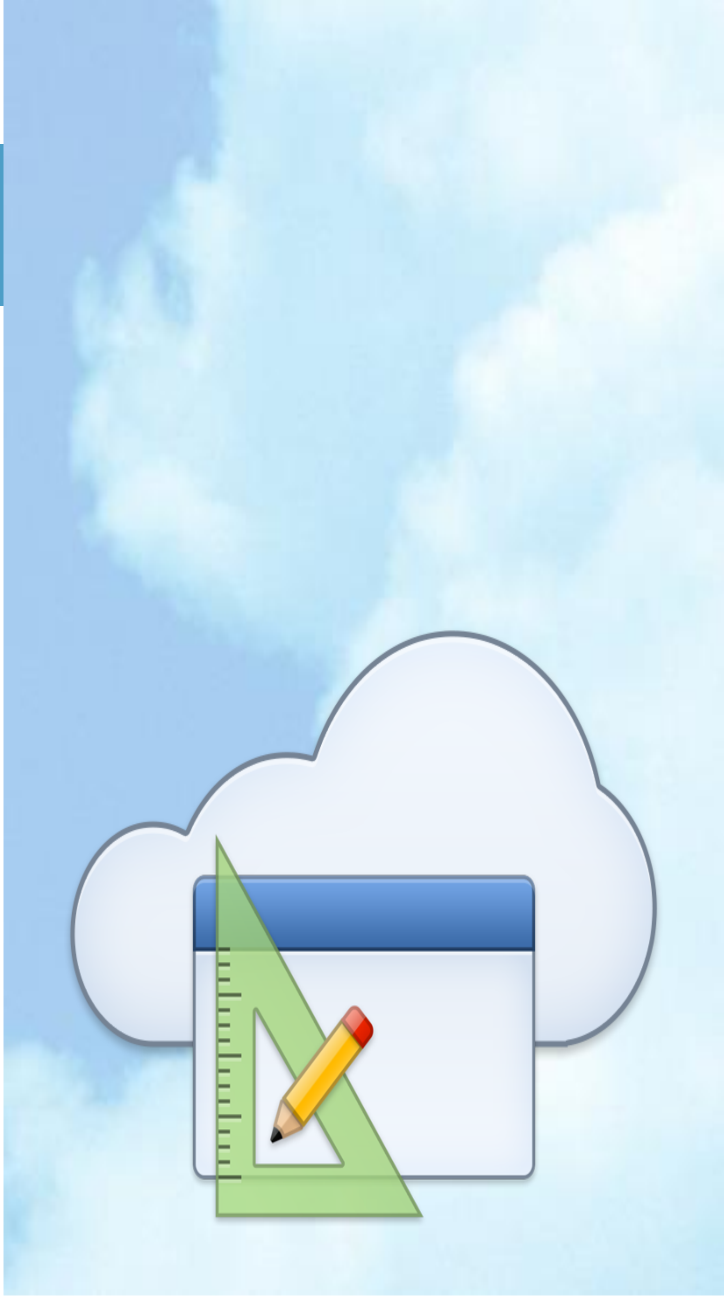
- Change My Password
- About Application Express
- Manage User Groups
- Create User
- Create Multiple Users
- Delete Multiple Users
- Manage Interactive Report

Service	
Workspace Name	DPEAKE
Database Applications	121
Websheet Applications	5
Packaged Applications	7

Users	
Users	20
Workspace Administrators	16
Application Developers	19
End Users	1

Activity	
Reporting Timeframe	3 hours
Page Events	46
Median Page Time	0.26
Distinct Applications	6

O-APEX features

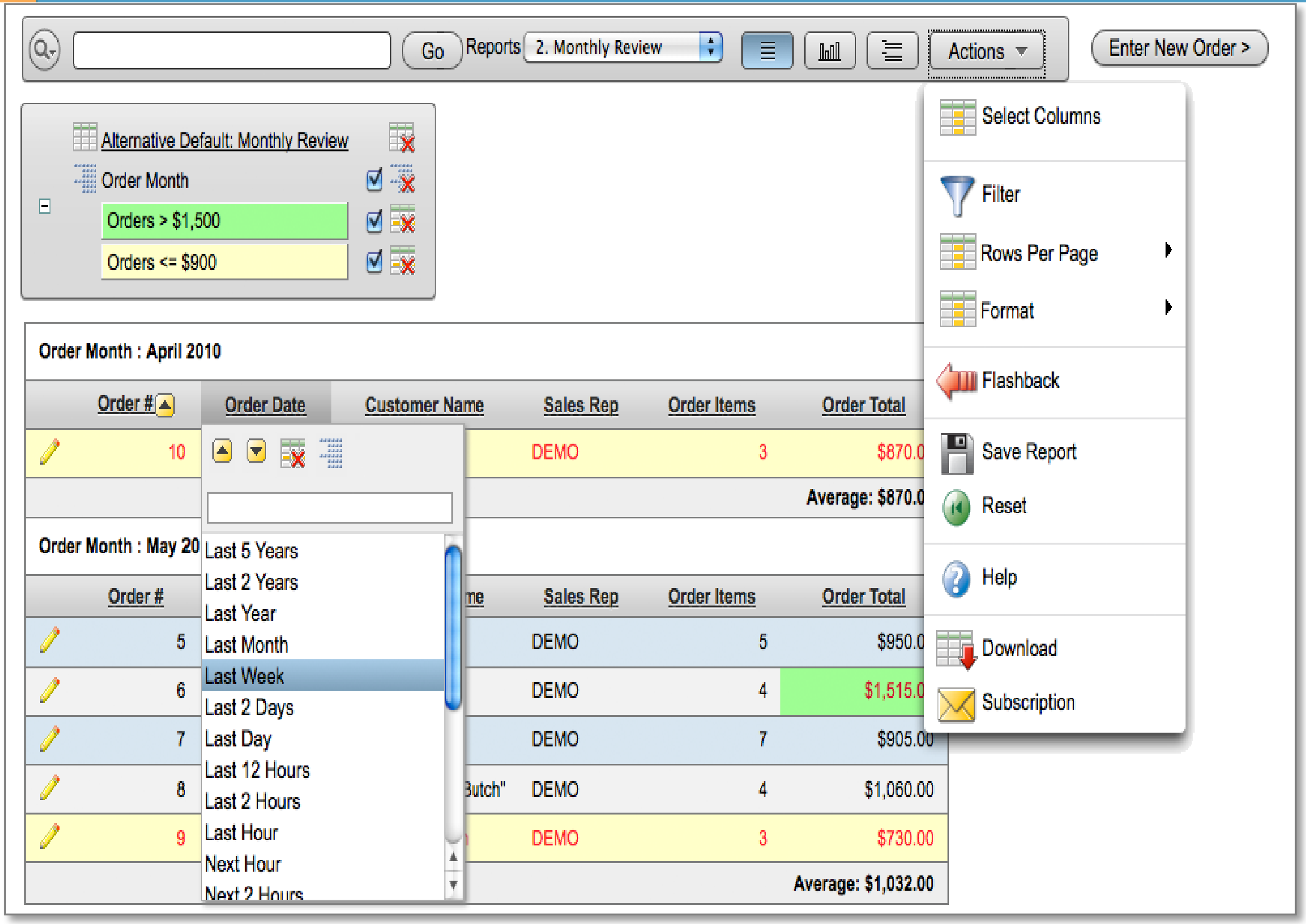


Oracle Application Express - Key Features

Declarative ca s >> Majority of otl Tools

Reports	Validations	Translation Services
Forms	Processes	Conditional Processing
Charts	Computations	Authentication
Calendar	Branches	Authorization
Templates	Web Services	Session State Management
Navigation	Email Services	Logging & Monitoring

Oracle Application Express - Features



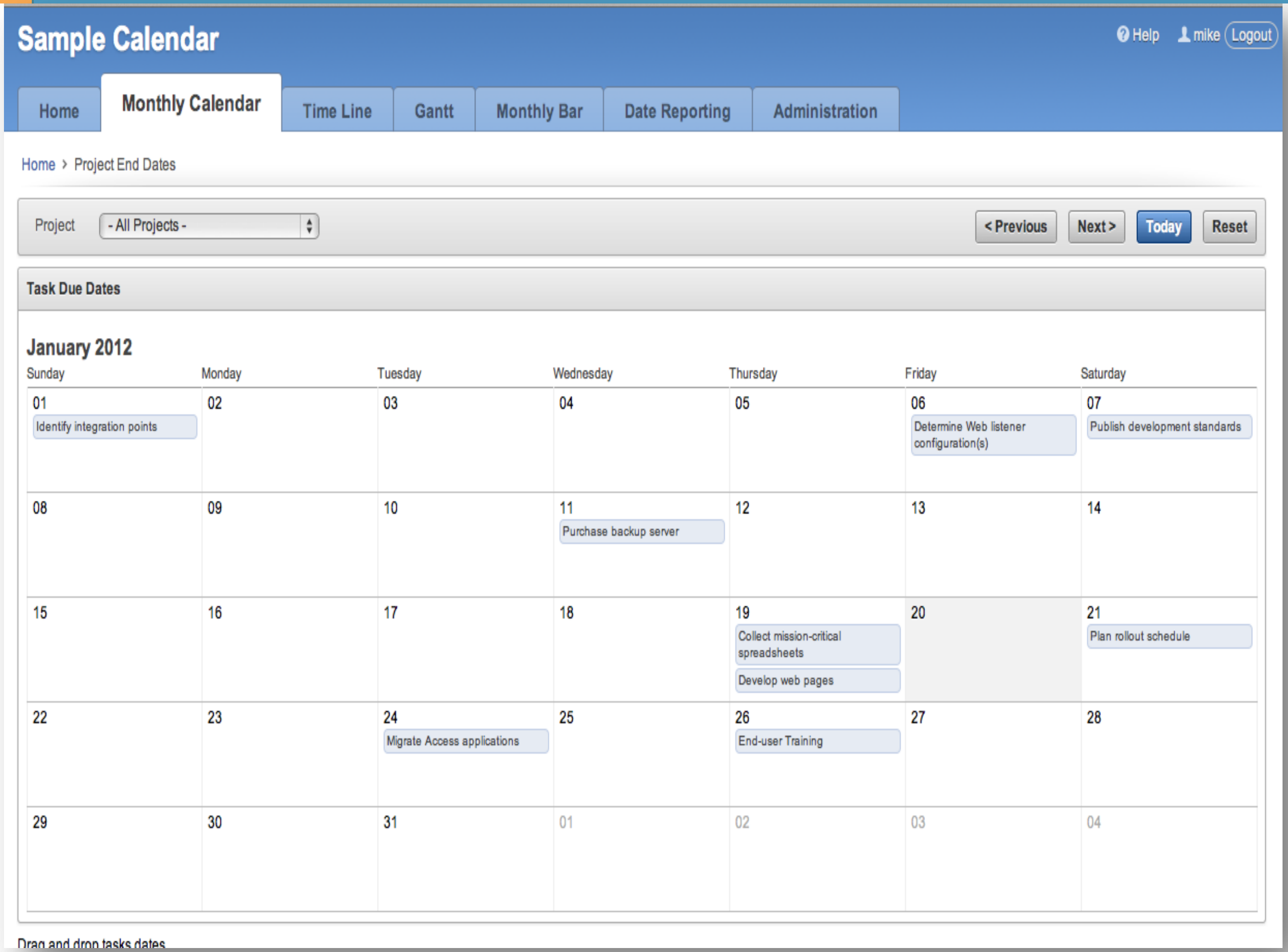
Order Month : April 2010

Order #	Order Date	Customer Name	Sales Rep	Order Items	Order Total
10			DEMO	3	$870.0
					Average: $870.0

Order Month : May 20

Order #		me	Sales Rep	Order Items	Order Total
5			DEMO	5	$950.0
6			DEMO	4	$1,515.0
7			DEMO	7	$905.00
8		Butch"	DEMO	4	$1,060.00
9			DEMO	3	$730.00
					Average: $1,032.00

Oracle Application Express - Features

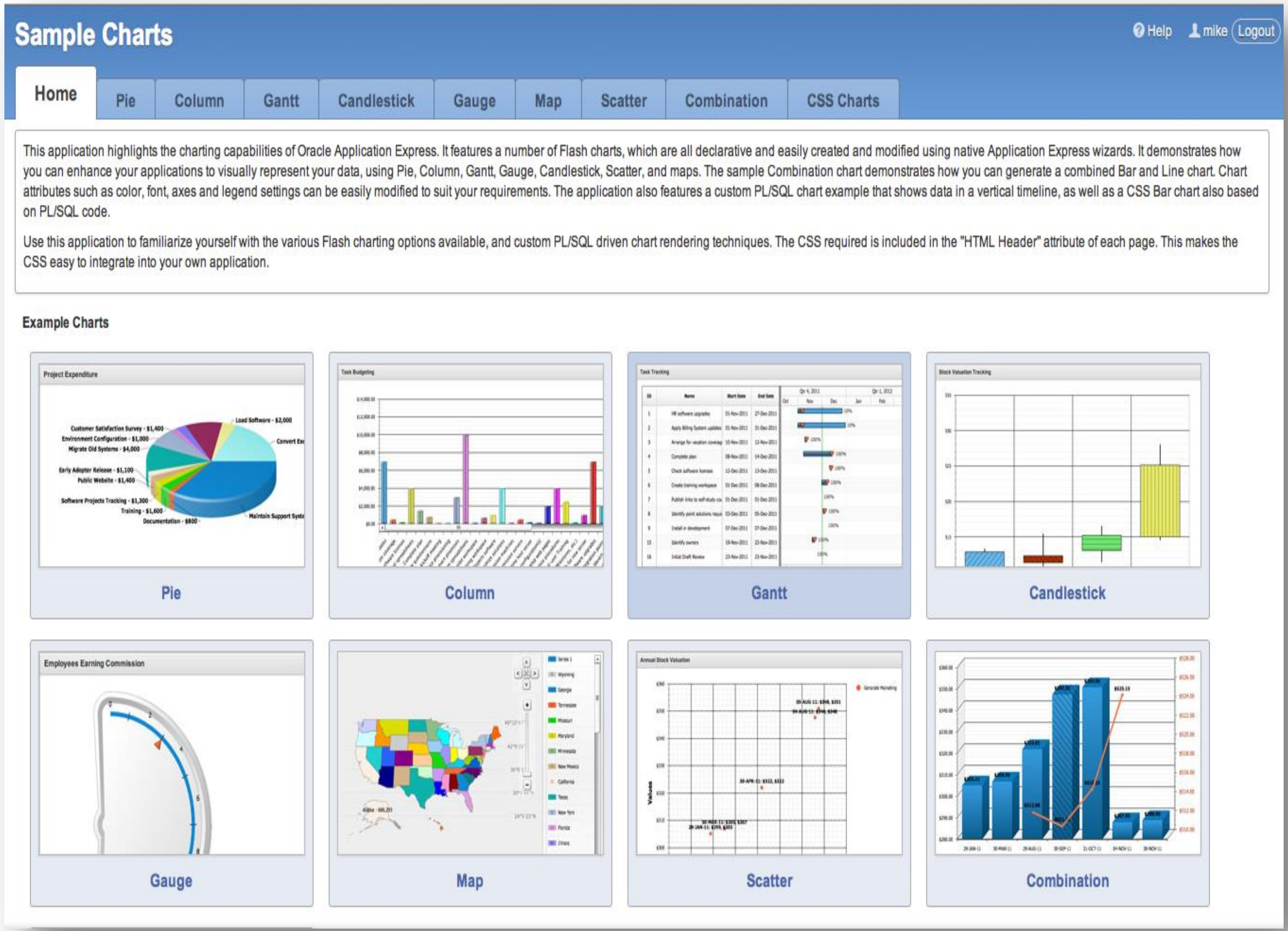


January 2012

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
01 Identify integration points	02	03	04	05	06 Determine Web listener configuration(s)	07 Publish development standards
08	09	10	11 Purchase backup server	12	13	14
15	16	17	18	19 Collect mission-critical spreadsheets; Develop web pages	20	21 Plan rollout schedule
22	23	24 Migrate Access applications	25	26 End-user Training	27	28
29	30	31	01	02	03	04

Drag and drop tasks dates

Oracle Application Express - Features

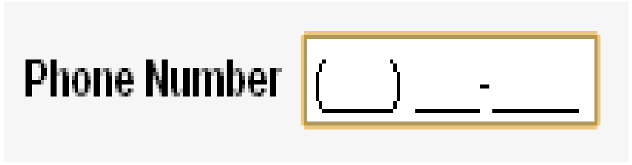


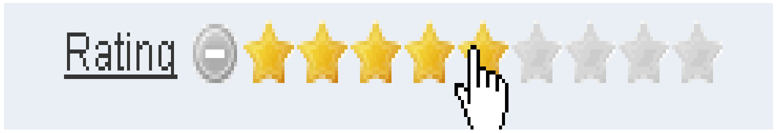
This application highlights the charting capabilities of Oracle Application Express. It features a number of Flash charts, which are all declarative and easily created and modified using native Application Express wizards. It demonstrates how you can enhance your applications to visually represent your data, using Pie, Column, Gantt, Gauge, Candlestick, Scatter, and maps. The sample Combination chart demonstrates how you can generate a combined Bar and Line chart. Chart attributes such as color, font, axes and legend settings can be easily modified to suit your requirements. The application also features a custom PL/SQL chart example that shows data in a vertical timeline, as well as a CSS Bar chart also based on PL/SQL code.

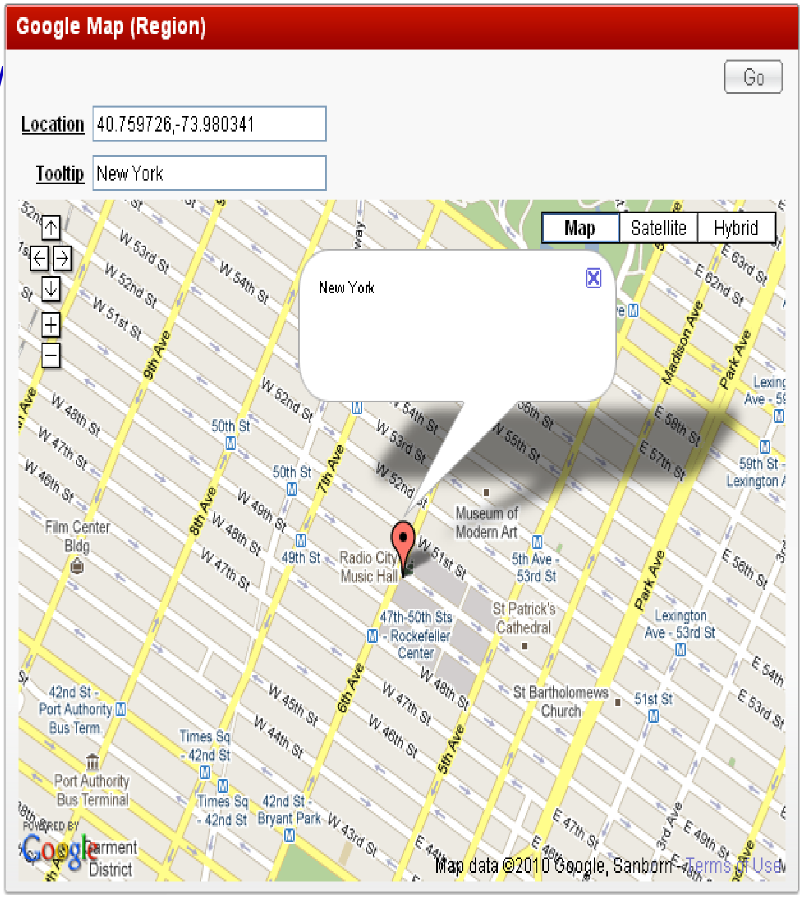
Use this application to familiarize yourself with the various Flash charting options available, and custom PL/SQL driven chart rendering techniques. The CSS required is included in the "HTML Header" attribute of each page. This makes the CSS easy to integrate into your own application.

Oracle Application Express - Features

Plug-Ins → Extend framework w






Google Map (Region)

Go

Location 40.759726,-73.980341

Tooltip New York