

Semester 1 Mid Term Review

Semester 1 Mid Term Review

This slide set contains the following topics:

Introduction to PL/SQL	Benefits of PL/SQL
Creating PL/SQL Blocks	Review of SQL SELECT Statements
Review of Single-Row Functions	Using Variables in PL/SQL
Recognizing PL/SQL Lexical Units	Recognizing Data Types
Using Scalar Data Types	Review of SQL Joins
Review of SQL Group Functions and Subqueries	Nested Blocks and Variable Scope
Writing PL/SQL Executable Statements	Review of SQL DML
Good Programming Practices	Manipulating Data in PL/SQL
Retrieving Data in PL/SQL	Conditional Control: IF Statements
Using Transaction Control Statements	Iterative Control: Basic Loops
Conditional Control: CASE Statements	Iterative Control: Nested Loops
Iterative Control: WHILE and FOR Loops	Using Explicit Cursor Attributes
Introduction to Explicit Cursors	Cursors with Parameters
Cursor FOR Loops	Using Multiple Cursors
Using Cursors for Update	

Introduction to PL/SQL

What is PL/SQL?

- Stands for Procedural Language extension to SQL
 - It allows basic program logic and control flow to be combined with SQL statements
- Is an Oracle proprietary programming language
 - It can be used only with an Oracle database or tool
- Is a procedural language
 - It produces a result when a series of instructions are followed
- Is a 3GL (third-generation programming language)
 - It is a “high-level” programming language



What is Structured Query Language (SQL)?

- The primary language used to access and modify data in a relational database
- A nonprocedural language
 - Also known as a "declarative language," it allows the programmer to focus on input and output rather than the program steps
- A 4GL (fourth-generation-programming language)
 - A language that is closer to natural language than a programming language; query languages are generally 4GL
- A common query language for many types of databases including Oracle
- A language standardized by the American National Standards Institute (ANSI)

 **PL/SQL extends SQL with Procedural Logic:**

```
DECLARE
  v_new_letter_grade varchar2(1);
  CURSOR c_enrollments IS
    SELECT stu_id, final_numeric_grade FROM enrollments WHERE class_id=1;
BEGIN
  FOR c1 IN c_enrollments
  LOOP
    IF c1.final_numeric_grade BETWEEN 66 and 75 THEN v_new_letter_grade := 'A';
    ELSIF c1.final_numeric_grade BETWEEN 56 AND 65 THEN v_new_letter_grade := 'B';
    ELSIF c1.final_numeric_grade BETWEEN 46 AND 55 THEN v_new_letter_grade := 'C';
    ELSIF c1.final_numeric_grade BETWEEN 36 AND 45 THEN v_new_letter_grade := 'D';
    ELSE
      v_new_letter_grade := 'F';
    END IF;
    UPDATE enrollments
      SET final_letter_grade=v_new_letter_grade WHERE class_id=1
      AND stu_id=c1.stu_id;
  END LOOP;
  COMMIT;
END;
```

Procedural Constructs:

We use PL/SQL to write the procedural code, and embed SQL data-accessing statements within the PL/SQL code.

- The code uses variables, cursors, and conditional logic.
- PL/SQL provides procedural constructs such as:
 - Variables, constants, and types
 - Control structures such as conditional statements and loops
 - Reusable program units that are written once and executed many times



Procedural Constructs:

```
DECLARE
...
BEGIN
  FOR c1 IN c_enrollments
  LOOP
    IF c1.final_numeric_grade BETWEEN 66 AND 75 THEN
      v_new_letter_grade := 'A';
    ELSIF c1.final_numeric_grade BETWEEN 56 AND 65 THEN
      v_new_letter_grade := 'B';
    ...
    ELSE
      v_new_letter_grade := 'F';
    END IF;
    UPDATE enrollments
      SET final_letter_grade=v_new_letter_grade
      WHERE class_id=1
      AND stu_id=c1.stu_id;
  END LOOP;
END;
```

Cursor (points to `c_enrollments`)

Iterative control (points to `LOOP`)

Conditional control (points to the `IF` block)

SQL (points to the `UPDATE` statement)

Variable (points to `v_new_letter_grade`)

Benefits of PL/SQL



Benefit 1: Integration of procedural constructs with SQL

The first and foremost advantage of PL/SQL is the integration of procedural constructs with SQL.

- SQL is a nonprocedural language. When you issue a SQL command, your command tells the database server what to do. However, you cannot specify how to do it.
- PL/SQL integrates control statements and conditional statements with SQL. This gives you better control of your SQL statements and their execution.

Benefit 2: Modularized program development

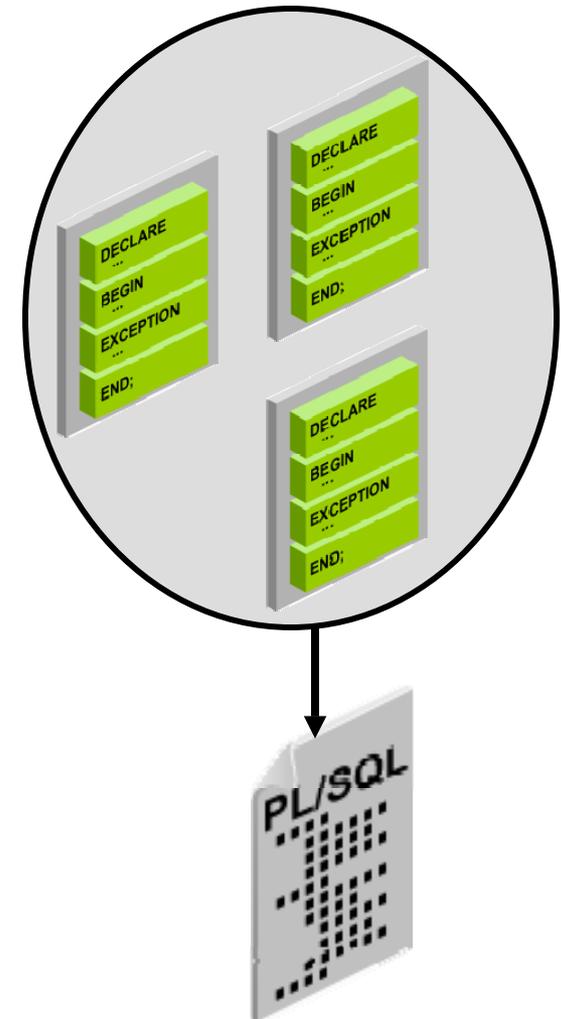
The basic unit in a PL/SQL program is a block. All PL/SQL programs consist of blocks. These blocks can be thought of as modules and can be “modularized” in a sequence or nested in other blocks.



Benefit 2: Modularized program development (continued)

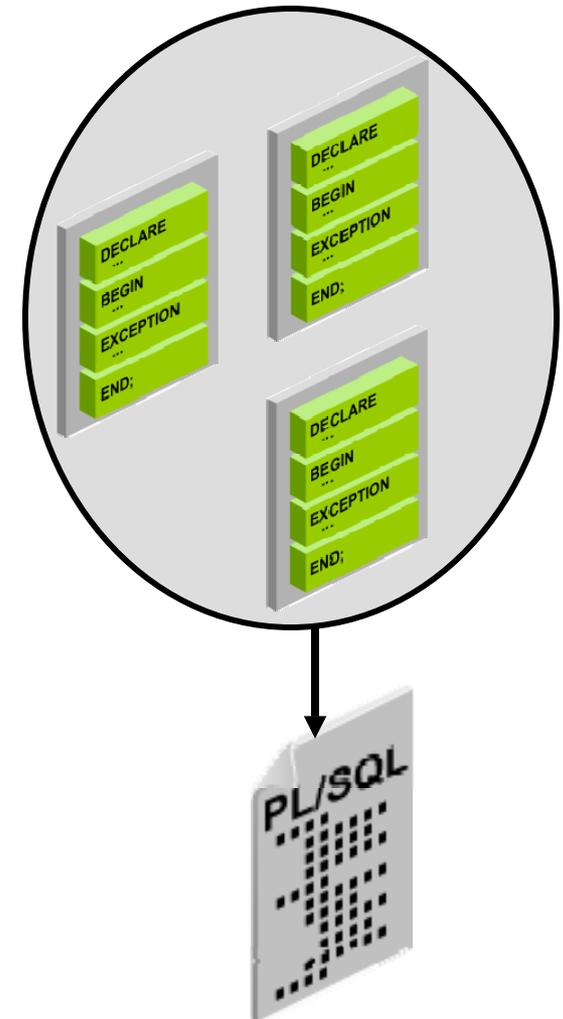
Modularized program development has the following advantages:

- You can group logically related statements within blocks.
- You can nest blocks inside other blocks to build powerful programs.



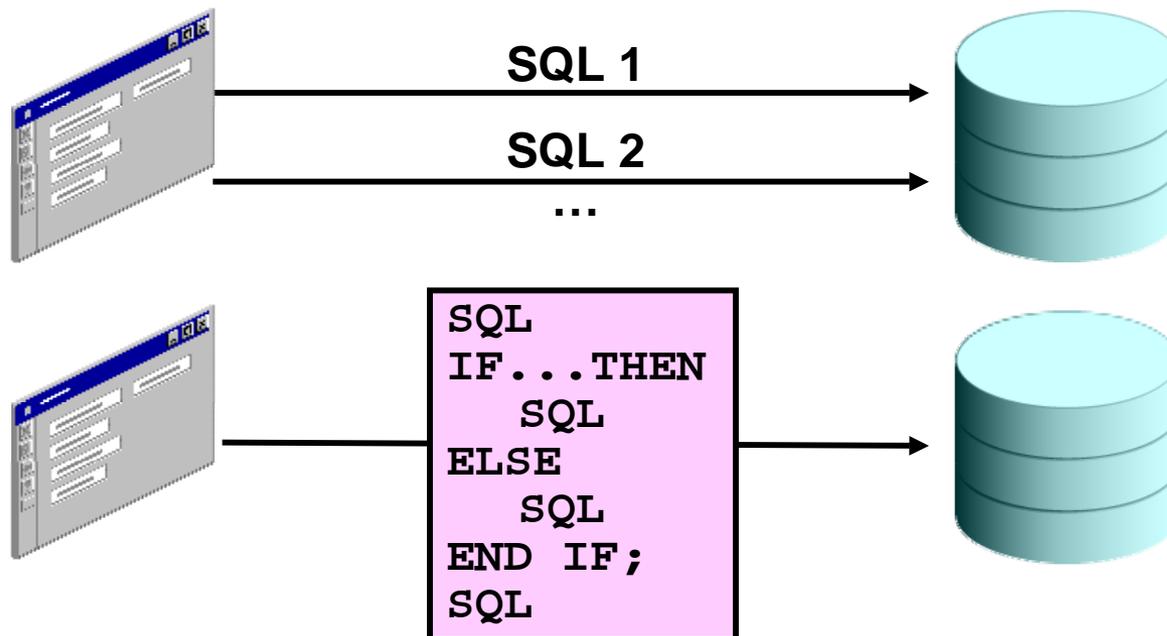
Benefit 2: Modularized program development (continued)

- You can break your application into smaller modules. If you are designing a complex application, PL/SQL allows you to break down the application into smaller, manageable, and logically related modules.
- You can easily read, maintain, and debug the code.



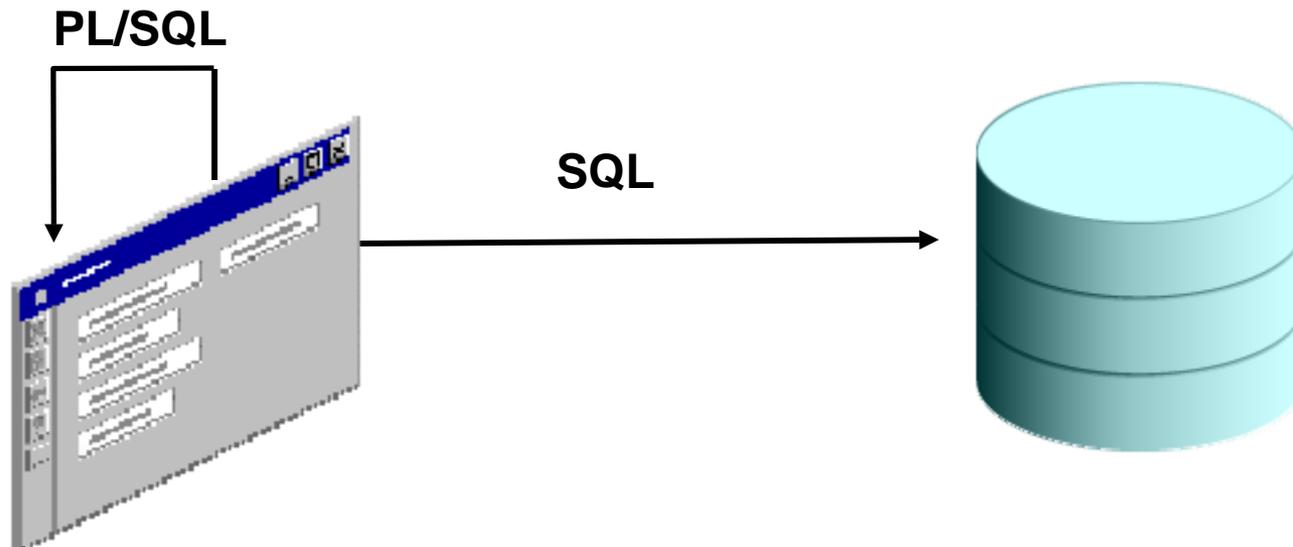
Benefit 3: Improved performance

PL/SQL allows you to logically combine multiple SQL statements as one unit or block. The application can send the entire block to the database instead of sending the SQL statements one at a time. This significantly reduces the number of database calls.



Benefit 4: Integration with Oracle tools

PL/SQL is integrated in Oracle tools such as Oracle Forms Developer, Oracle Report Builder, and Application Express.



Benefit 5: Portability

PL/SQL programs can run anywhere an Oracle server runs, irrespective of the operating system and the platform. PL/SQL programs do not need to be tailored for different operating systems and platforms.

You can write portable program packages and create libraries that can be reused on Oracle databases in different environments.



Linux



HP Tru64



IBM z/OS

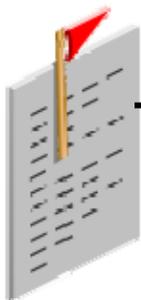


Solaris

Benefit 6: Exception handling

An exception is an error that occurs in the database or in a user's program during runtime. Examples of errors include: hardware or network failures, application logic errors, data integrity errors, and so on. You can prepare for errors by writing exception handling code. Exception handling code tells your program what to do in the event of an exception.

PL/SQL allows you to handle database and program exceptions efficiently. You can define separate blocks for dealing with exceptions.



If there is no data found then ...

If too many rows are returned then...

If an invalid number is returned then ...

Creating PL/SQL Blocks



PL/SQL Block Structure

A PL/SQL block consists of three sections:

Declarative (optional): The declarative section begins with the keyword `DECLARE` and ends when your executable section starts.

Executable (required): The executable section begins with the keyword `BEGIN` and ends with `END`. Observe that `END` is terminated with a semicolon. The executable section of a PL/SQL block can in turn include any number of nested PL/SQL blocks.

Exception handling (optional): The exception section is nested within the executable section. This section begins with the keyword `EXCEPTION`.

PL/SQL Block Structure (continued)

DECLARE (Optional)

Variables, cursors, user-defined exceptions

BEGIN (Mandatory)

- SQL statements
- PL/SQL statements

EXCEPTION (Optional)

Actions to perform
when errors occur

END; (Mandatory)

In a PL/SQL block, the keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not terminated by a semicolon. However, the keyword `END`, all SQL statements, and PL/SQL statements must be terminated with a semicolon.



 **PL/SQL Block Structure (continued)**

Section	Description	Inclusion
Declarative (DECLARE)	Contains declarations of all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and exception sections.	Optional
Executable (BEGIN ... END;)	Contains SQL statements to retrieve data from the database and PL/SQL statements to manipulate data in the block. Must contain at least one statement.	Mandatory
Exception (EXCEPTION)	Specifies the actions to perform when errors and abnormal conditions arise in the executable section.	Optional

Block Types

A PL/SQL program comprises one or more blocks. These blocks can be entirely separate or nested within another.

There are three types of blocks that can make up a PL/SQL program. They are:

- Anonymous blocks
- Procedures  Subprograms
- Functions  Subprograms





Anonymous Blocks

Anonymous blocks:

- Unnamed blocks
- Not stored in the database
- Declared inline at the point in an application where they are executed
- Compiled each time the application is executed
- Passed to the PL/SQL engine for execution at run time
- Cannot be invoked or called because they do not have names and do not exist after they are executed

```
DECLARE ]  
  
BEGIN  
  --statements  
  
[ EXCEPTION ]  
  
END;
```



Examples of Anonymous Blocks

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('PL/SQL is easy!');
END;
```

```
DECLARE
    v_date DATE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE(v_date);
END;
```

```
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT last_name INTO v_lname
    FROM employees WHERE job_id = 'ST_CLERK';
    DBMS_OUTPUT.PUT_LINE('The last name of the ST_CLERK is :
' || v_lname);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE (' Your select statement retrieved multiple
rows. Consider using a cursor. ');
END;
```



Subprograms

Subprograms:

- Are named PL/SQL blocks
- Are stored in the database
- Can be invoked whenever you want to depending on your application
- Can be declared as procedures or as functions
 - Procedure: performs an action
 - Function: computes and returns a value

```
PROCEDURE name
IS
BEGIN
    --statements

[ EXCEPTION ]

END;
```

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

END;
```

 **Examples of Subprograms:**

1. Procedure to print the current date

```
CREATE PROCEDURE print_date IS
  v_date VARCHAR2(30);
BEGIN
  SELECT TO_CHAR(SYSDATE, 'Mon DD, YYYY')
    INTO v_date
    FROM DUAL;
  DBMS_OUTPUT.PUT_LINE(v_date);
END;
```

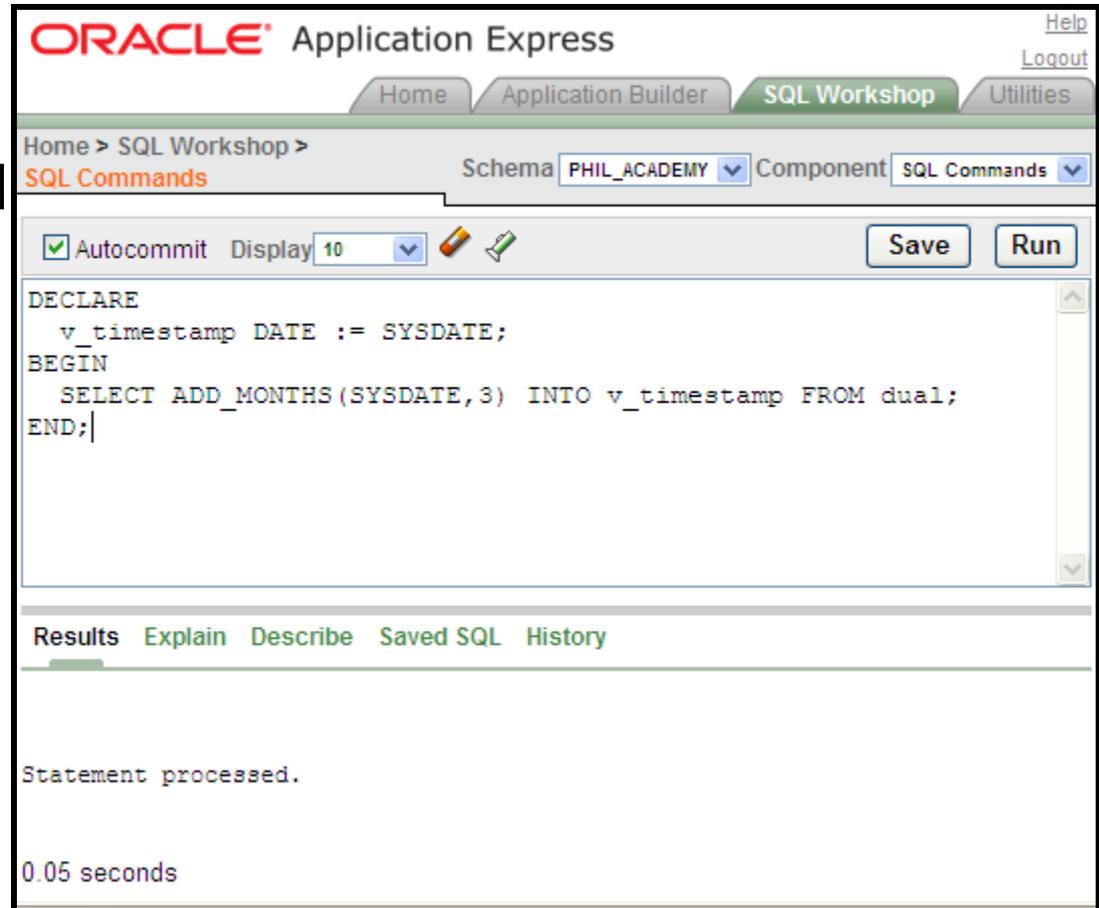
2. Function to return the number of characters in a string

```
CREATE FUNCTION num_characters (p_string IN
VARCHAR2)
  RETURN INTEGER IS
  v_num_characters INTEGER;
BEGIN
  SELECT LENGTH(p_string) INTO v_num_characters
    FROM dual;
  RETURN v_num_characters;
END;
```

SQL Commands

You can use **SQL Commands** to enter and run a single SQL statement or a single PL/SQL block.

A SQL script can contain one or more SQL statements and/or PL/SQL blocks. Use **SQL Scripts** to enter and run multi-statement scripts.



The screenshot displays the Oracle Application Express SQL Workshop interface. The breadcrumb navigation shows 'Home > SQL Workshop > SQL Commands'. The schema is set to 'PHIL_ACADEMY' and the component is 'SQL Commands'. The 'Autocommit' checkbox is checked, and the 'Display' dropdown is set to '10'. The 'Save' and 'Run' buttons are visible. The SQL editor contains the following code:

```
DECLARE
  v_timestamp DATE := SYSDATE;
BEGIN
  SELECT ADD_MONTHS(SYSDATE,3) INTO v_timestamp FROM dual;
END;
```

Below the editor, the 'Results' tab is selected, showing the output: 'Statement processed.' and '0.05 seconds'.

Review of SQL SELECT Statements



Selecting Data

`SELECT` is the keyword that retrieves column(s) from a table.

The `FROM` clause specifies the table name(s).

`SELECT * FROM tablename` retrieves all the data in a table.

`SELECT <column list> FROM tablename` retrieves the columns specified.

The `WHERE` clause specifies a condition that restricts the rows returned by the `SELECT` statement.

 **Sorting**

ORDER BY specifies the display sequence of the result. The keywords ASC or DESC may be added after the column name to specify ascending or descending sequence.

```
SELECT country_name
FROM wf_countries
ORDER BY country_name;
```

COUNTRY_NAME
Anguilla
Antarctica
Antigua and Barbuda
Arab Republic of Egypt
Argentine Republic
Aruba
Bailiwick of Guernsey
Bailiwick of Jersey
Barbados
Belize
More than 10 rows available. Increase rows selector to view more rows.



Calculations

This example uses the multiplication operator to calculate the new area of Benin, if a land reclamation project increased its area by 2 percent.

```
SELECT country_name, area, area * 1.02
FROM wf_countries
WHERE country_id = 229;
```

COUNTRY_NAME	AREA	AREA*1.02
Republic of Benin	112620	114872.4



Column aliases

This example uses an alias to display the calculated value as "New Area".

```
SELECT country_name, area, area * 1.02 "New Area"  
FROM wf_countries  
WHERE country_id = 229;
```

COUNTRY_NAME	AREA	AREA*1.02
Republic of Benin	112620	114872.4



Concatenation

Concatenation means to connect or link together in a series. The concatenation operator is || (2 vertical bars sometimes referred to as “pipes”).

A literal value is a character, a number, or a date that is included in the `SELECT` list and that is not a column name or a column alias. Literal values are often used with concatenation to create readable text output.

```
SELECT country_name || ' has an area of ' || area  
as "Readable Text"  
FROM wf_countries;
```

Readable Text
Aruba has an area of 193
Antigua and Barbuda has an area of 443
United Arab Emirates has an area of 82880
Islamic Republic of Afghanistan has an area of 647500
Peoples Democratic Republic of Algeria has an area of 2381740
Republic of Azerbaijan has an area of 86600
Republic of Albania has an area of 28748

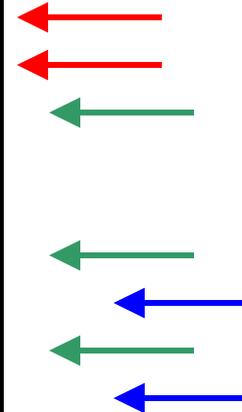


DISTINCT

The `DISTINCT` keyword is used to eliminate duplicate rows from the output of a SQL statement. This example returns all the region IDs from the `WF_COUNTRIES` table. Notice the duplicates.

```
SELECT region_id
FROM wf_countries;
```

REGION_ID
29
29
145
34
15
145
39
145
39
18
More than 10 rows available. Increase rows selector to view more rows.





DISTINCT

The `DISTINCT` keyword is used to eliminate duplicate rows from the output of a SQL statement. This example eliminates the duplicates.

```
SELECT DISTINCT region_id  
FROM wf_countries;
```

REGION_ID
34
151
30
29
155
13
11
21
14
5
More than 10 rows available. Increase rows selector to view more rows.



BETWEEN...AND

The BETWEEN...AND operator is used to select and display rows based on a range of values. The BETWEEN...AND condition is specified in the WHERE clause and the results include the boundary values if present.

```
SELECT country_name, coastline
FROM wf_countries
WHERE coastline BETWEEN 500 AND 550;
```

COUNTRY_NAME	COASTLINE
Republic of Cote d Ivoire	515
Republic of Kenya	536
Republic of Latvia	531
Republic of Ghana	539
Paracel Islands	518
Commonwealth of Puerto Rico	501
Republic of Senegal	531

**IN**

The `IN` condition is used to test whether a value is in a specified set of values.

This example selects countries that are in region 2 or 3.

```
SELECT region_id, country_name
FROM wf_countries
WHERE region_id IN (5,9);
```

REGION ID	COUNTRY NAME
9	Territory of American Samoa
5	Argentine Republic
9	Commonwealth of Australia
9	Antarctica
5	Republic of Bolivia
9	Solomon Islands
5	Federative Republic of Brazil
5	Republic of Chile
5	Republic of Colombia
9	Commonwealth of the Northern Mariana Islands
More than 10 rows available. Increase rows selector to view more rows.	



LIKE

The `LIKE` condition allows you to select rows that match either literal strings or number patterns.

The `%` and the underscore (`_`) are wildcard characters that can be used to construct a search string. The `%` symbol is used to represent any sequence of zero or more characters. The underscore (`_`) symbol is used to represent a single character.

```
SELECT country_name,  
       national_holiday_name  
FROM wf_countries  
WHERE national_holiday_name  
       LIKE '%Independence%';
```

COUNTRY_NAME	NATIONAL_HOLIDAY_NAME
Antigua and Barbuda	Independence Day (National Day)
United Arab Emirates	Independence Day
Islamic Republic of Afghanistan	Independence Day
Republic of Albania	Independence Day
Republic of Armenia	Independence Day
Republic of Angola	Independence Day
Barbados	Independence Day
Republic of Botswana	Independence Day
Commonwealth of The Bahamas	Independence Day
Peoples Republic of Bangladesh	Independence Day
More than 10 rows available. Increase rows selector to view more rows.	

Review of SQL Single-Row Functions



Case Manipulation Functions

Case manipulation functions used in a `SELECT` statement temporarily convert character data to a specified case.

`LOWER` converts alpha characters to lowercase.

```
SELECT country_id, country_name, area
FROM wf_countries
WHERE LOWER(country_name) = 'kingdom of tonga';
```

COUNTRY_ID	COUNTRY_NAME	AREA
676	Kingdom of Tonga	748



Case Manipulation Functions

UPPER converts alpha characters to uppercase.

```
SELECT country_id, country_name, area
FROM wf_countries
WHERE UPPER(country_name) = 'KINGDOM OF TONGA';
```

INITCAP converts alpha character values to uppercase for the first letter of each word.

```
SELECT country_id, country_name, area
FROM wf_countries
WHERE INITCAP (country_name) = 'Kingdom Of Tonga';
```

Character Manipulation Functions

Character manipulation functions temporarily convert character data to different values.

CONCAT joins two values together.

```
SELECT CONCAT (country_name,internet_extension)
           "Country and extension"
FROM wf_countries WHERE country_id = 229;
```

Country and extension
Republic of Benin.bj



Character Manipulation Functions

SUBSTR extracts a string of a determined length.

```
SELECT SUBSTR(country_name,3,3)
FROM wf_countries WHERE country_id = 229;
```

SUBSTR(COUNTRY_NAME,3,3)
pub

LENGTH shows the length of a string as a number value.

```
SELECT LENGTH(country_name )
FROM wf_countries WHERE country_id = 229;
```

LENGTH(COUNTRY_NAME)
17



Number Functions

Number functions temporarily convert number data to different values.

ROUND: Used to round numbers to a specified number of decimal places.

```
SELECT country_id, median_age, ROUND(median_age,1)
FROM wf_countries WHERE country_id = 20;
```

COUNTRY_ID	MEDIAN_AGE	ROUND(MEDIAN_AGE,1)
20	24	24



Number Functions

TRUNC: Used to terminate the column, expression, or value to a specified number of decimal places

```
SELECT TRUNC(999.128,2) FROM dual;
```

TRUNC(999.128,2)
999.12

MOD: Used to return the remainder when one number is divided by another

```
SELECT country_id,population, MOD(population,2)  
FROM wf_countries WHERE country_id = 3;
```

COUNTRY_ID	POPULATION	MOD(POPULATION,2)
3	15233244	0



Conversion Functions

TO_CHAR converts dates stored in a database to a format specified by you. The syntax is:

```
TO_CHAR (date, 'format model you specify')
```

Example:

```
SELECT TO_CHAR(SYSDATE, 'Month ddth, yyyy') as TODAY  
FROM dual;
```

TODAY
November 30 th , 2006



Conversion Functions

TO_CHAR converts columns of number data to a desired format.
The syntax is:

```
TO_CHAR (number, 'format model you specify')
```

Example:

```
SELECT country_id, TO_CHAR(population, '99,999,999,999')  
FROM wf_countries;
```

COUNTRY_ID	TO_CHAR(POPULATION, '99,999,999,999')
297	71,891
1268	69,108
971	2,602,713
93	31,056,997
213	32,930,091
994	7,961,619



Conversion Functions

TO_DATE converts a character string to a specified date format. The syntax is:

```
TO_DATE('character string', 'format model')
```

Example:

```
SELECT TO_DATE('January 1, 2006', 'Month DD, RRRR')  
       AS "New Year"  
FROM dual;
```

New Year
01-JAN-06



Conversion Functions

TO_NUMBER converts a character string to a number. The syntax is:

```
TO_NUMBER(character string, 'format model')
```

Example:

```
SELECT TO_NUMBER('95.5', '999.99')  
       AS converted  
FROM   dual;
```

CONVERTED
95.5



Date Functions

`SYSDATE` is a date function that returns the current database server date and time.

Example:

```
SELECT SYSDATE+1 as tomorrow
FROM dual;
```

TOMORROW
01-DEC-06



Date Functions

MONTHS_BETWEEN returns the number of months between two dates.

Example:

```
SELECT country_name "Country",
       date_of_independence "Independence Day",
       TO_CHAR(MONTHS_BETWEEN(SYSDATE,
                             date_of_independence), '999,999,999.99')
       AS "Months Since"
FROM wf_countries
WHERE country_id = 229;
```

Country	Independence Day	Months Since
Republic of Benin	1-Aug-1960	555.97



Date Functions

ADD_MONTHS increments a date by calendar months

Example:

```
SELECT ADD_MONTHS(SYSDATE, 120) "10 years from  
today"  
FROM dual;
```

10 yrs from today
30-NOV-16



General Functions

NVL converts a null value to a date, a character, or a number.

The syntax is:

```
NVL(value that may contain a null,  
value to replace the null)
```



General Functions

NVL examples:

```
SELECT currency_name, comments
FROM wf_currencies
WHERE currency_code = 'AUD';
```

CURRENCY_NAME	COMMENTS
Australian dollar	-

```
SELECT currency_name,
       NVL(comments, 'No comment') comments
FROM wf_currencies
WHERE currency_code = 'AUD';
```

CURRENCY_NAME	COMMENTS
Australian dollar	No comment

General Functions

NULLIF compares two functions. If they are equal, the function returns null. If they are not equal, the function returns the first expression. The syntax is:

```
NULLIF(expression 1, expression 2)
```

```
SELECT country_translated_name "Country Name Trans",
       country_name "Country Name",
       NULLIF(country_translated_name, country_name)
              "nullif returns"
FROM wf_countries;
```

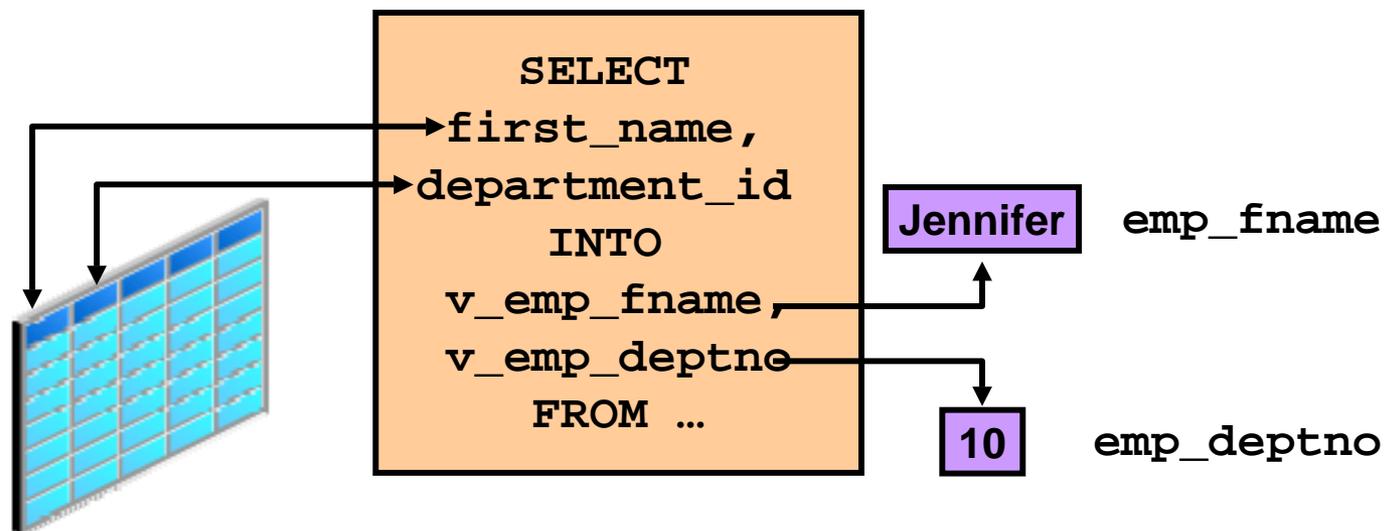
Country Name Trans	Country Name	nullif returns
-	Aruba	-
-	Antiqua and Barbuda	-
Al Imarat al Arabiyah al Muttahidah	United Arab Emirates	Al Imarat al Arabiyah al Muttahidah
Afghanistan	Islamic Republic of Afghanistan	Afghanistan
Algeria	Peoples Democratic Republic of Algeria	Algeria
Azerbaijan	Republic of Azerbaijan	Azerbaijan

Using Variables in PL/SQL

Use of Variables

Variables can be used for:

- Temporary storage of data
- Manipulation of stored values
- Reusability





Handling Variables in PL/SQL

Variables are:

- Declared and initialized in the declarative section
- Used and assigned new values in the executable section
- Passed as parameters to PL/SQL subprograms
- Assigned to hold the output of a PL/SQL subprogram

Declaring and Initializing PL/SQL Variables

- All PL/SQL variables must be declared in the declaration section before referencing them in the PL/SQL block.
- The purpose of a declaration is to allocate storage space for a value, specify its data type, and name the storage location so that you can reference it.
- Variables can be declared in the declarative part of any PL/SQL block, subprogram, or package.





Declaring and Initializing Variables:

Syntax:

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

- *identifier* is the name of the variable
- CONSTANT constrains the variable so that its value cannot change; constants must be initialized
- *datatype* is a scalar, composite, reference, or LOB data type
- NOT NULL constrains the variable so that it must contain a value (NOT NULL variables must be initialized.)
- *Expr* can be a literal, another variable, or an expression involving operators and functions



Declaring and Initializing Variables:

Examples

```
DECLARE
  v_emp_hiredate      DATE;
  v_emp_deptno       NUMBER(2) NOT NULL := 10;
  v_location         VARCHAR2(13) := 'Atlanta';
  c_comm             CONSTANT NUMBER := 1400;
  v_population       INTEGER;
  v_book_type        VARCHAR2(20) DEFAULT 'fiction';
  v_artist_name      VARCHAR2(50);
  v_firstname        VARCHAR2(20) := 'Rajiv';
  v_lastname         VARCHAR2(20) DEFAULT 'Kumar';
  c_display_no       CONSTANT PLS_INTEGER := 20;
```

...

Assigning Values in the Executable Section

After a variable is declared, you can assign it a value and use it in the executable section of a PL/SQL block.

```
DECLARE
  v_myname VARCHAR2(20);
BEGIN
  DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
  v_myname := 'John';
  DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
END;
```

The output is:

```
My name is:
My name is:  John

Statement process.
```

Assigning Values in the Executable Section

In this block, the variable `v_myname` is declared and initialized in the declarative section. `v_myname` will hold the value `John` after initialization. Note that this value is manipulated in the executable section of the block.

```
DECLARE
  v_myname VARCHAR2(20) := 'John';
BEGIN
  v_myname := 'Steven';
  DBMS_OUTPUT.PUT_LINE('My name is: ' || v_myname);
END;
```

The output is:

```
My name is: Steven
```

```
Statement processed.
```

Passing Variables as Parameters to PL/SQL Subprograms

Parameters are values passed to a program by the user or by another program to customize the program.

In PL/SQL, subprograms can take parameters. You can pass variables as parameters to procedures and functions.

In the following example, the parameter `v_date` is being passed to the procedure `PUT_LINE`, which is part of the package, `DBMS_OUTPUT`.

```
DECLARE
    v_date VARCHAR2(30);
BEGIN
    SELECT TO_CHAR(SYSDATE) INTO v_date
        FROM dual;
    DBMS_OUTPUT.PUT_LINE(v_date);
END;
```



Assigning Variables to PL/SQL Subprogram Output

Variables can be used to hold the value that is returned by a function.

```
FUNCTION num_characters (p_string IN VARCHAR2) RETURN INTEGER
IS
  v_num_characters INTEGER;
BEGIN
  SELECT LENGTH(p_string) INTO v_num_characters
  FROM dual;
  RETURN v_num_characters;
END;
```

```
DECLARE
  v_length_of_string INTEGER;
BEGIN
  v_length_of_string := num_characters('Oracle Corporation');
  DBMS_OUTPUT.PUT_LINE(v_length_of_string);
END;
```

Recognizing PL/SQL Lexical Units



Lexical Units in a PL/SQL Block

Lexical units:

- Are the building blocks of any PL/SQL block
- Are sequences of characters including letters, digits, tabs, returns, and symbols
- Can be classified as:
 - Identifiers
 - Reserved words
 - Delimiters
 - Literals
 - Comments

 **Identifiers**

An identifier is the name given to a PL/SQL object, including any of the following:

Procedure	Function	Variable
Exception	Constant	Package
Record	PL/SQL table	Cursor



Identifiers

The identifiers in the following PL/SQL code are highlighted.

```
PROCEDURE print_date IS  
  
    v_date VARCHAR2(30);  
  
BEGIN  
  
    SELECT TO_CHAR(SYSDATE, 'Mon DD, YYYY')  
        INTO v_date  
        FROM dual;  
    DBMS_OUTPUT.PUT_LINE(v_date);  
  
END;
```

Key:  Procedure  Variable  Reserved word



Properties of an Identifier

Identifiers:

- Are up to 30 characters in length
- Must start with a letter
- Can include \$ (dollar sign), _ (underscore), and # (pound sign)
- Cannot contain spaces



Valid and Invalid Identifiers

Examples of valid identifiers:

First_Name	LastName	address_1
ID#	Total_\$	primary_department_contact

Examples of invalid identifiers:

First Name	Contains a space
Last-Name	Contains invalid "-"
1st_address_line	Begins with a number
Total_%	Contains invalid "%"
primary_building_department_contact	More than 30 characters



Reserved Words

Reserved words are words that have special meaning to the Oracle database.

Reserved words cannot be used as identifiers in a PL/SQL program.

This is a partial list of reserved words.

ALL	CREATE	FROM	MODIFY	SELECT
ALTER	DATE	GROUP	NOT	SYNONYM
AND	DEFAULT	HAVING	NULL	SYSDATE
ANY	DELETE	IN	NUMBER	TABLE
AS	DESC	INDEX	OR	THEN
ASC	DISTINCT	INSERT	ORDER	UPDATE
BETWEEN	DROP	INTEGER	RENAME	VALUES
CHAR	ELSE	INTO	ROW	VARCHAR2
COLUMN	EXISTS	IS	ROWID	VIEW
COMMENT	FOR	LIKE	ROWNUM	WHERE



Reserved Words

What happens when you try to use a reserved word as an identifier in a PL/SQL program?

```
DECLARE
  date DATE;
BEGIN
  SELECT ADD_MONTHS(SYSDATE,3) INTO date
  FROM dual;
END;
```

```
ORA-06550:  line 4, column 37:
PL/SQL:  ORA-00936:  missing expression
ORA-06550:  line 4, column 3:
PL/SQL:  SQL Statement ignored
2.          date DATE;
3.  BEGIN
4.          SELECT ADD_MONTHS(SYSDATE,3) INTO date
5.          FROM DUAL;
6.  END;
```

 **Delimiters are symbols that have special meaning to the Oracle database.**

Simple delimiters

Symbol	Meaning
+	Addition operator
-	Subtraction/negation operator
*	Multiplication operator
/	Division operator
=	Equality operator
'	Character string delimiter
;	Statement terminator

Compound delimiters

Symbol	Meaning
<>	Inequality operator
!=	Inequality operator
	Concatenation operator
--	Single-line comment indicator
/*	Beginning comment delimiter
*/	Ending comment delimiter
:=	Assignment operator

 **Literals**

A literal is an explicit numeric, character string, date, or Boolean value that is not represented by an identifier.

Literals are classified as:

- Character (also known as string literals)
- Numeric
- Boolean



Character Literals

- Include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.
- Have the data type `CHAR` and must be enclosed in single quotation marks.
- May be composed of zero or more characters from the PL/SQL character set.
- Are case sensitive and, therefore, `PL/SQL` is not equivalent to `pl/sql`.

```
v_first_name := 'John';  
v_classroom  := '12C';  
v_date_today := '20-MAY-2005';
```

Numeric Literals

- Are values that represent an integer or real value
- Examples :

```
v_elevation      := 428;  
v_order_subtotal := 1025.69;  
v_distance_sun_to_centauri := 4.3E13;
```

Boolean Literals

- Are not surrounded by quotes.
- TRUE, FALSE, and NULL are Boolean literals or keywords.

```
v_new_customer      := FALSE;  
v_paid_in_full      := TRUE;  
v_high_school_diploma := NULL;  
v_island            := FALSE;
```



Comments

- Explain what a piece of code is trying to achieve.
- Are extremely valuable for code readability and maintenance.
- Are good programming practice.
- Are not interpreted or processed by the PL/SQL compiler.
- Two dashes -- are used for commenting a single line.
- /* */ is used for commenting multiple lines.

```
DECLARE
  v_annual_sal NUMBER (9,2);
BEGIN
  -- Begin the executable section

  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_annual_sal := v_monthly_sal * 12;
END;
  -- This is the end of the block
```

Recognizing Data Types



PL/SQL Data Types

- A data type specifies a storage format, constraints, and a valid range of values.
- PL/SQL supports five categories of data type:
 - Scalar: Hold a single value
 - Composite: Contain internal elements that are either scalar (record) or composite (record and table)
 - Large object (LOB): Hold values, called locators, that specify the location of large objects (such as graphic images) that are stored out of line
 - Reference: Hold values, called pointers, that point to a storage location
 - Object: A schema object with a name, attributes and methods. An object data type is similar to the class mechanism supported by C++ and Java.

Scalar Data Types

Scalar data types:

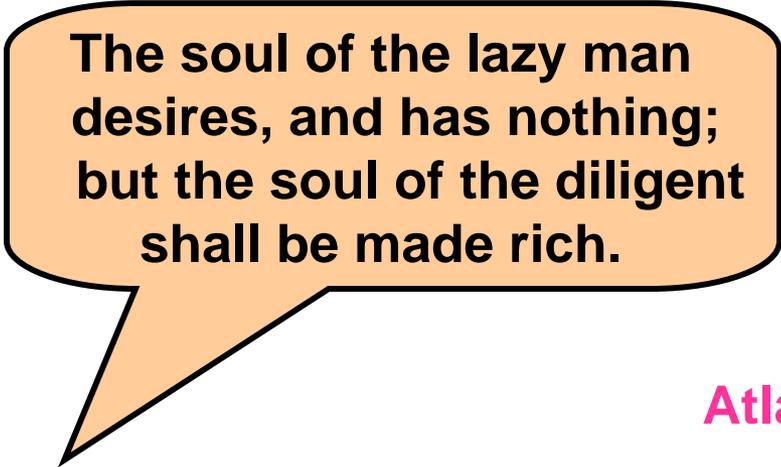
- Hold a single value
- Have no internal components
- Can be classified into four

categories:

- Character
- Number
- Date
- Boolean

TRUE

25-JAN-01



**The soul of the lazy man
desires, and has nothing;
but the soul of the diligent
shall be made rich.**

Atlanta

256120.08

 **Scalar Data Types: Character (or String)**

CHAR [(<i>maximum_length</i>)]	Base type for fixed-length character data up to 32,767 bytes. The default length is 1.
VARCHAR2 (<i>maximum_length</i>)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
LONG	Character data of variable length
LONG RAW	Raw binary data of variable length (not interpreted by PL/SQL)

Scalar Data Types: Boolean

BOOLEAN	Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, or NULL
---------	--



Scalar Data Types: Number

NUMBER [(<i>precision</i> , <i>scale</i>)]	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 and BINARY_INTEGER values require less storage and are faster than NUMBER values.
BINARY_FLOAT BINARY_DOUBLE	New data types introduced in Oracle Database 10g. They represent floating-point number in the IEEE 754 format. BINARY_FLOAT requires 5 bytes to store the value and BINARY_DOUBLE requires 9 bytes.



Scalar Data Types: Date

DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 4712 B.C. and A.D. 9999.
TIMESTAMP	The TIMESTAMP data type, which extends the DATE data type, stores the year, month, day, hour, minute, second, and fraction of seconds
TIMESTAMP WITH TIME ZONE	The TIMESTAMP WITH TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement—that is, the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time

 **Scalar Data Types: Date (continued)**

TIMESTAMP WITH LOCAL TIME ZONE	This data type differs from <code>TIMESTAMP WITH TIME ZONE</code> in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, Oracle server returns the value in your local session time zone.
INTERVAL YEAR TO MONTH	You use the <code>INTERVAL YEAR TO MONTH</code> data type to store and manipulate intervals of years and months.
INTERVAL DAY TO SECOND	You use the <code>INTERVAL DAY TO SECOND</code> data type to store and manipulate intervals of days, hours, minutes, and seconds.

Composite Data Types

A scalar type has no internal components. A composite type has internal components that can be manipulated individually.

Composite data types (also known as collections) are of the following types: RECORD, TABLE, NESTED TABLE, VARRAY

PL/SQL record structure

TRUE	23-DEC-98	ATLANTA	
------	-----------	---------	---

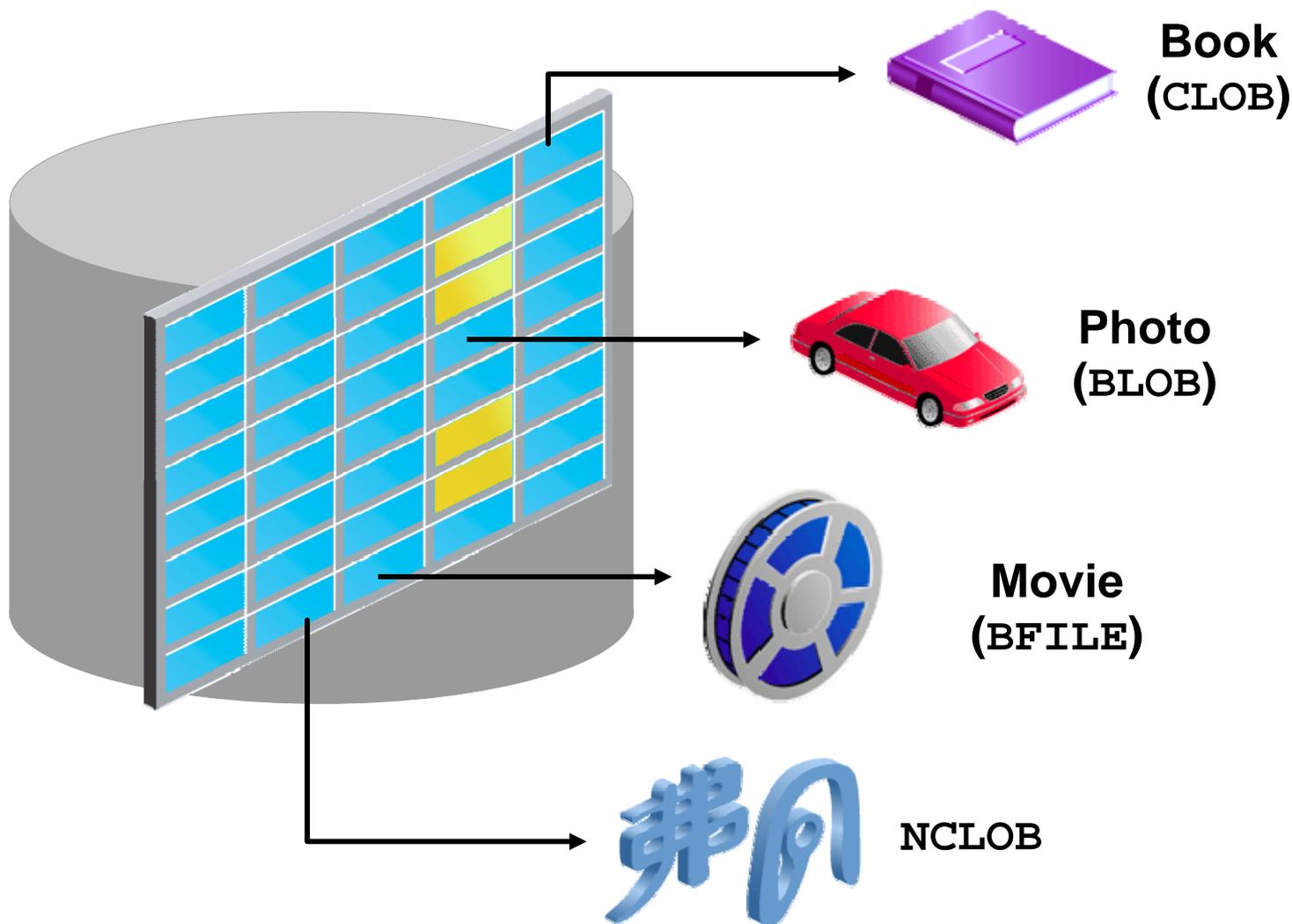
PL/SQL table structure

PLS_INTEGER	1	SMITH	VARCHAR2
	2	JONES	
	3	NANCY	
	4	TIM	

 **LOB Data Type**

- Large objects (LOBs) are meant to store a large amount of data.
- Database columns can be of the LOB category.
- There are several categories of LOB data types:
 - character large object (CLOB)
 - binary large object (BLOB)
 - binary file (BFILE)
 - national language character large object (NCLOB)
- Enable you to store blocks of unstructured data up to 4 gigabytes in size.
- Enable efficient, random, piecewise access to the data and can be attributes of an object type.

LOB Data Type Columns



Using Scalar Data Types

Declaring Character Variables

Character data types include CHAR, VARCHAR2 and LONG.

```
DECLARE
  v_emp_job          VARCHAR2(9);
  v_order_no        VARCHAR2(6);
  v_product_id      VARCHAR2(10);
  v_rpt_body_part   LONG;
```

Declaring Number Variables

Number data types include NUMBER, PLS_INTEGER, BINARY_INTEGER, and BINARY_FLOAT.

```
DECLARE
  v_dept_total_sal  NUMBER(9,2) := 0;
  count_loop       BINARY_INTEGER := 0;
  c_tax_rate        CONSTANT NUMBER(3,2) := 8.25;
```



Declaring Date Variables

Date data types include DATE, TIMESTAMP, and TIMESTAMP WITH TIMEZONE.

```
DECLARE
  v_orderdate          DATE := SYSDATE + 7;
  v_natl_holiday      DATE;
  v_web_sign_on_date  TIMESTAMP;
  ...
```

Declaring Boolean Variables

A Boolean variable stores one of the three possible values used for logical calculations: TRUE, FALSE, or NULL.

```
DECLARE
  v_valid          BOOLEAN NOT NULL := TRUE;
  v_is_found       BOOLEAN := FALSE;
  v_underage       BOOLEAN;
```

- Conditional expressions use the logical operators AND, OR, or NOT to check the variable values.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

Guidelines for Declaring and Initializing PL/SQL Variables

- Use meaningful names and follow naming conventions.
- Declare one identifier per line for better readability and code maintenance.
- Use the `NOT NULL` constraint when the variable must hold a value.
- Avoid using column names as identifiers.

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



Anchoring Variables with the %TYPE Attribute

Rather than hard-coding the data type and precision of a variable, you can use the %TYPE attribute to declare a variable according to another previously declared variable or database column. The %TYPE attribute is most often used when the value stored in the variable will be derived from a table in the database.

```
CREATE TABLE myemps (  
    emp_name          VARCHAR2(6),  
    emp_salary        NUMBER(6,2));  
  
DECLARE  
    v_emp_salary      myemps.emp_salary%TYPE;  
BEGIN  
    SELECT emp_salary INTO v_emp_salary  
    FROM myemps WHERE emp_name = 'Smith';  
END;
```



Advantages of the %TYPE Attribute

- You can avoid errors caused by data type mismatch or wrong precision.
- You need not change the variable declaration if the column definition changes. That is, if you have already declared some variables for a particular table without using the %TYPE attribute, then the PL/SQL block may return errors if the column for which the variable declared is altered.
- When you use the %TYPE attribute, PL/SQL determines the data type and size of the variable when the block is compiled. This ensures that such a variable is always compatible with the column that is used to populate it.

Review of SQL Joins



Equijoin

An equijoin combines rows that have equal values for the specified columns. This is sometimes called a "simple" join.

```
SELECT r.region_name, c.country_name, c.airports
FROM wf_world_regions r, wf_countries c
WHERE r.region_id = c.region_id
ORDER BY c.country_name;
```

REGION_NAME	COUNTRY_NAME	AIRPORTS
Caribbean	Anguilla	3
Oceania	Antarctica	28
Caribbean	Antigua and Barbuda	3
Northern Africa	Arab Republic of Egypt	87
South America	Argentine Republic	1333
Caribbean	Aruba	1



Nonequijoin

A nonequijoin combines tables that do not have an exact matching columns. This join is often used when a column value in table A falls between a range of values specified by two columns in table B.

Examine the following tables:

```
SELECT employee_id,  
        last_name, salary  
FROM employees;
```

EMPLOYEE_ID	LAST_NAME	SALARY
100	King	24000
101	Kochlar	17000
102	De Haan	17000
103	Hunold	9000
104	Ernst	6000
107	Lorentz	4200

```
SELECT *  
FROM job_grades;
```

GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000



Nonequijoin

In this example, the grade for each student is matched up with the equivalent grade code.

```
SELECT e.employee_id, e.last_name, j.grade_level
       FROM employees e, job_grades j
       WHERE e.salary BETWEEN j.lowest_sal AND j.highest_sal;
```

EMPLOYEE_ID	LAST_NAME	GRADE_LEVEL
144	Vargas	A
143	Matos	A
142	Davies	B
141	Rajs	B
107	Lorentz	B
200	Whalen	B



Outer Join

An outer join combines rows that have equivalent values for the specified columns plus those rows in one of the tables that have no matching value in the other table. To indicate which table may have missing data, use a plus sign (+) after the table's column name in the query.

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_name, d.department_id
FROM   employees e, departments d
WHERE  e.department_id = d.department_id (+)
ORDER BY NVL(e.department_id,1) ASC;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME	DEPARTMENT_ID
178	Grant	-	-	-
200	Whalen	10	Administration	10
202	Fay	20	Marketing	20
201	Hartstein	20	Marketing	20
124	Mourgos	50	Shipping	50
141	Rajs	50	Shipping	50
142	Davies	50	Shipping	50



Cartesian Product

When a join query does not specify a condition in the `WHERE` clause, the result is a Cartesian product. This combines each row of one table with each row of the other resulting in many rows. The query below produces $244 \times 637 = 155,428$ rows!

```
SELECT country_name, language_id  
FROM wf_countries, wf_spoken_languages;
```

COUNTRY_NAME	LANGUAGE_ID
United Arab Emirates	460
United Arab Emirates	1850
United Arab Emirates	460
United Arab Emirates	560
United Arab Emirates	900
United Arab Emirates	1650
United Arab Emirates	1650
United Arab Emirates	0
United Arab Emirates	460

Review of SQL Group Functions and Subqueries



Group Functions

These functions operate on a whole table or on a specific grouping of rows to return one result.

```
SELECT TO_CHAR(AVG(population), '9,999,999,999.99') average
FROM wf_countries;
```

AVERAGE
27,130,848.07

```
SELECT COUNT(country_id) "Number of Countries"
FROM wf_countries;
```

```
SELECT COUNT(*) "Number of Countries"
FROM wf_countries;
```

Number of Countries
244



Group Functions

```
SELECT MIN(lowest_elevation)"All time low"  
FROM wf_countries;
```

All time low
-2555

```
SELECT MAX(highest_elevation)"All time high"  
FROM wf_countries;
```

All time low
8850

```
SELECT TO_CHAR(SUM(area), '9,999,999,999.99') "Total area"  
FROM wf_countries;
```

Total area
148,148,433.00



GROUP BY

Use the `GROUP BY` clause to divide the rows in a table into smaller groups. You can then use the group functions to return summary information for each group.

The `WHERE` clause first excludes rows. The remaining data is divided into groups, and group functions are applied.

```
SELECT region_id, COUNT(country_id)
FROM wf_countries
WHERE region_id < 15
GROUP BY region_id;
```

REGION_ID	COUNT(COUNTRY_ID)
5	15
9	28
11	21
13	8
14	7



HAVING

Use the `HAVING` clause to restrict groups. In a query that uses a `GROUP BY` and `HAVING` clause, the rows are first grouped, group functions are applied, and then only those groups matching the `HAVING` clause are displayed.

```
SELECT region_id, COUNT(country_id)
FROM wf_countries
WHERE region_id < 15
GROUP BY region_id
HAVING COUNT(country_id) < 20
ORDER BY region_id DESC;
```

REGION_ID	COUNT(COUNTRY_ID)
14	7
13	8
5	15



Subqueries

A subquery is a `SELECT` statement that is embedded in a clause of another `SQL` statement.

Subqueries can be placed in a number of `SQL` clauses, including the `WHERE` clause, the `HAVING` clause, and the `FROM` clause.

Single-row subqueries use single-row operators (`>`, `=`, `>=`, `<` `<>`, and `<=`) and return only one row from the subquery.

```
SELECT country_name,  
TO_CHAR(area, '999,999,999,999') area  
FROM wf_countries  
WHERE area >  
      (SELECT area  
       FROM wf_countries  
       WHERE country_name =  
        'Federative Republic of Brazil')  
ORDER BY country_name;
```

COUNTRY_NAME	AREA
Antarctica	14,000,000
Canada	9,984,670
People Republic of China	9,984,670
Russian Federation	17,075,200
United States of America	9,631,420



Group Functions and Subqueries

Group functions can be used in subqueries because they return a single row.

```
SELECT country_name
FROM wf_countries
WHERE lowest_elevation =
      (SELECT MIN (lowest_elevation)
       FROM wf_countries);
```

COUNTRY_NAME
Antarctica



Group Functions

This example shows that subqueries can be used with joins and other WHERE conditions.

```
SELECT country_name, population
FROM wf_countries
WHERE population =
  (SELECT MAX(population)
   FROM wf_countries c, wf_world_regions wr
   WHERE c.region_id = wr.region_id
   AND wr.region_name = 'Oceania');
```

COUNTRY_NAME	POPULATION
Commonwealth of Australia	20264082



Multiple-Row Subqueries

Use multiple-row operators (`IN`, `ANY`, and `ALL`) to return more than one row from the inner query.

The `IN` operator is used when the outer query `WHERE` clause is designed to restrict rows based on a list of values returned from the inner query.

```
SELECT country_name , population, airports
FROM wf_countries
WHERE country_id IN
      (SELECT country_id
       FROM wf_countries WHERE airports >1);
```

COUNTRY_NAME	POPULATION	AIRPORTS
United Arab Emirates	2602713	35
Republic of Azerbaijan	7961619	45
Republic of Armenia	2976372	16
Commonwealth of Australia	20264082	450
Republic of Austria	8192880	55
Antarctica	0	28
Republic of Botswana	1639833	85

 **ANY Operator**

The `ANY` operator is used when the outer query `WHERE` clause is designed to restrict rows based on any value returned from the inner query.

```
SELECT country_name, population, area
FROM wf_countries
WHERE country_id = ANY
      (SELECT country_id
       FROM wf_countries
       WHERE area <1000);
```

COUNTRY_NAME	POPULATION	AREA
Territory of Cocos (Keeling) Islands	574	14
Jay Mayen	-	373
Principality of Liechtenstein	33987	160
Principality of Monaco	32543	2
Republic of Maldives	359008	300

 **ALL Operator**

The `ALL` operator is used when the outer query `WHERE` clause is designed to restrict rows based on all values returned from the inner query.

```
SELECT country_name
FROM wf_countries c, wf_world_regions wr
WHERE c.region_id = wr.region_id
AND region_name > ALL
  (SELECT region_name
   FROM wf_world_regions
   WHERE UPPER(region_name) LIKE 'A%');
```

COUNTRY_NAME
United Arab Emirates
Republic of Azerbaijan
Republic of Armenia

Writing PL/SQL Executable Statements



Assigning New Values to Variables

- Character and date literals must be enclosed in single quotation marks.

```
v_name      := 'Henderson';  
v_start_date := '12-DEC-2005';
```

- Wrapped statements can continue over several lines.

```
v_quote := 'Now is the time for all good  
men and women to come to the aid of  
their country';
```

- Numbers can be simple values or scientific notation.

```
v_my_integer := 100;  
v_my_sci_not := 2E5;
```

(2E5 meaning 2×10 to the power of 5 = 200,000)



SQL Functions in PL/SQL

You are already familiar with functions in SQL statements. For example:

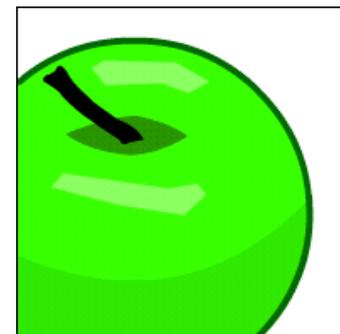
```
SELECT last_name, MONTHS_BETWEEN(SYSDATE, hiredate)
FROM employees;
```

These functions can also be used in PL/SQL procedural statements. For example:

```
DECLARE
    v_last_day DATE;
BEGIN
    v_last_day := LAST_DAY(SYSDATE);
    DBMS_OUTPUT.PUT_LINE(v_last_day);
END;
```

SQL Functions in PL/SQL

- Available in both SQL and procedural statements:
 - Single-row character
 - Single-row number
 - Date
 - Data type conversion
 - Miscellaneous functions
- Not available in procedural statements, but are available in SQL statements:
 - DECODE
 - Group functions





Character Functions

ASCII	LENGTH	RPAD
CHR	LOWER	RTRIM
CONCAT	LPAD	SUBSTR
INITCAP	LTRIM	TRIM
INSTR	REPLACE	UPPER

```
v_desc_size          INTEGER(5);  
v_prod_description  VARCHAR2(70):='You can use this  
product with your radios for higher frequency';  
  
-- get the length of the string in prod_description  
v_desc_size:= LENGTH(v_prod_description);
```

```
v_capitol_name:= UPPER(v_capitol_name);
```

```
v_emp_name:= v_first_name || ' ' || v_last_name;
```



Number Functions:

ABS	EXP	ROUND
ACOS	LN	SIGN
ASIN	LOG	SIN
ATAN	MOD	TAN
COS	POWER	TRUNC

```
DECLARE
v_my_num BINARY_INTEGER := -56664;
BEGIN
  DBMS_OUTPUT.PUT_LINE(SIGN(v_my_num));
END;
```

```
DECLARE
  v_median_age NUMBER(6,2);
BEGIN
  SELECT median_age INTO v_median_age
  FROM wf_countries WHERE country_id=27;
  DBMS_OUTPUT.PUT_LINE(ROUND(v_median_age,0));
END;
```



Date Functions:

ADD_MONTHS	MONTHS_BETWEEN
CURRENT_DATE	ROUND
CURRENT_TIMESTAMP	SYSDATE
LAST_DAY	TRUNC

```
DECLARE
  v_new_date    DATE;
  v_num_months  NUMBER := 6;
BEGIN
  v_new_date := ADD_MONTHS(SYSDATE, v_num_months);
  DBMS_OUTPUT.PUT_LINE(v_new_date);
END;
```

```
DECLARE
  v_no_months  PLS_INTEGER:=0;
BEGIN
  v_no_months := MONTHS_BETWEEN('31-JAN-06', '31-MAY-05');
  DBMS_OUTPUT.PUT_LINE(v_no_months);
END;
```

Data Type Conversion

In any programming language, converting one data type to another is a common requirement. PL/SQL can handle such conversions with scalar data types.

Implicit Conversions

In implicit conversions, PL/SQL attempts to convert data types dynamically if they are mixed in a statement. Implicit conversions can happen between many types in PL/SQL as illustrated by the following chart.

	DATE	LONG	NUMBER	PLS_INTEGER	VARCHAR2
DATE	N/A	X			X
LONG		N/A			X
NUMBER		X	N/A	X	X
PLS_INTEGER		X	X	N/A	X
VARCHAR2	X	X	X	X	N/A



Example of Implicit Conversion

Consider the following example:

```
DECLARE
  v_salary          NUMBER(6) := 6000;
  v_sal_increase    VARCHAR2(5) := '1000';
  v_total_salary    v_salary%TYPE;
BEGIN
  v_total_salary := v_salary + v_sal_increase;
  DBMS_OUTPUT.PUT_LINE(v_total_salary);
END;
```

In the example shown, the variable `v_sal_increase` is of type `VARCHAR2`. While calculating the total salary, PL/SQL will first convert `v_sal_increase` to `NUMBER` and then perform the operation. The result of the operation will then be of the `NUMBER` type. Also, the output will be converted to a string.

Drawbacks of Implicit Conversions

Although at first glance, implicit conversions may seem useful, there are several drawbacks:

- Implicit conversions can be slower.
- When you use implicit conversions, you lose control over your program because you are making an assumption about how Oracle will handle the data. If Oracle changes the conversion rules, then your code may be affected.
- Implicit conversion rules are dependent upon the environment in which you are running. For example, the date format varies depending on the language setting and installation type. Code that uses implicit conversion may not run on a different server or in a different language.
- Code that uses implicit conversion is harder to read and understand.

 **Drawbacks of Implicit Conversions continued**

It is the programmer's responsibility to ensure that values are convertible. For instance, PL/SQL can convert the CHAR value '02-JUN-92' to a DATE value but cannot convert the CHAR value 'Yesterday' to a DATE value. Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value.

Valid?	Statement
Yes	<code>v_new_date DATE := '02-JUN-1992';</code>
No	<code>v_new_date DATE := 'Yesterday';</code>
Yes	<code>v_my_number NUMBER := '123';</code>
No	<code>v_my_number NUMBER := 'abc';</code>



Explicit Conversions

Explicit conversions convert values from one data type to another by using built-in functions. Examples of conversion functions include:

TO_NUMBER ()	ROWIDTONCHAR ()
TO_CHAR ()	HEXTORAW ()
TO_CLOB ()	RAWTOHEX ()
CHARTOROWID ()	RAWTONHEX ()
ROWIDTOCHAR ()	TO_DATE ()

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE, 'Month YYYY'));
END;
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(TO_DATE('April-1999', 'Month YYYY'));
END;
```

 Examples of Explicit Conversions (continued)

```
DECLARE
  v_a VARCHAR2(10) := '-123456';
  v_b VARCHAR2(10) := '+987654';
  v_c PLS_INTEGER;
BEGIN
  v_c := TO_NUMBER(v_a) + TO_NUMBER(v_b);
  DBMS_OUTPUT.PUT_LINE(v_c);
END;
```

```
v_date_of_joining DATE:= 'February 02,2000';
```

← **FAILS**

```
v_date_of_joining DATE:= TO_DATE('February 02,2000',
'Month DD, YYYY');
```



Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control the order of operations

- Exponential operator (**)



**Same as
in SQL**

Operators in PL/SQL

The operations within an expression are performed in a particular order depending on their precedence (priority). The following table shows the default order of operations from high priority to low priority:

Operator	Operation
**	Exponentiation
+, -	Identity, negation
*, /	Multiplication, division
+, -,	Addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Inclusion



Operators in PL/SQL

Examples:

- Increment the counter for a loop.

```
v_loop_count := v_loop_count + 1;
```

- Set the value of a Boolean flag.

```
v_good_sal := v_sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value.

```
v_valid      := (v_empno IS NOT NULL);
```

Nested Blocks and Variable Scope



Nested Blocks

The example shown in the slide has an outer (parent) block (illustrated in blue) and a nested (child) block (illustrated in red). The variable `v_outer_variable` is declared in the outer block and the variable `v_inner_variable` is declared in the inner block.

```
DECLARE
  v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Variable Scope

The scope of a variable is the block or blocks in which the variable is accessible, ie it can be named and used. In PL/SQL, a variable's scope is the block in which it is declared plus all blocks nested within the declaring block.

What are the scopes of the two variables declared in this example?

```
DECLARE
  v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Examine the following code. What is the scope of each of the variables?

```
DECLARE
  v_father_name    VARCHAR2(20):='Patrick';
  v_date_of_birth  DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name    VARCHAR2(20):='Mike';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father's Name: ' || v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child's Name: ' || v_child_name);
  END;
  DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
END;
```

Local and Global Variables

Variables declared in a PL/SQL block are considered local to that block and global to all its subblocks. `v_outer_variable` is local to the outer block but global to the inner block. When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name. If there are no similarly named variables, PL/SQL looks for the variable in the outer block.

```
DECLARE
  v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Local and Global Variables continued

The `v_inner_variable` variable is local to the inner block and is not global because the inner block does not have any nested blocks. This variable can be accessed only within the inner block. If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks. PL/SQL does not look downward in the child blocks.

```
DECLARE
  v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



Variable Scope

The variables `v_father_name` and `v_date_of_birth` are declared in the outer block. They are local to the outer block and global to the inner block. Their scope includes both blocks.

```
DECLARE
  v_father_name    VARCHAR2(20):='Patrick';
  v_date_of_birth  DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name    VARCHAR2(20):='Mike';
    ...
```

The variable `v_child_name` is declared in the inner (nested) block. This variable is accessible only within the nested block and is not accessible in the outer block.



Variable Naming

You cannot declare two variables with the same name in the same block. However, you can declare variables with the same name in two different blocks (nested blocks). It is not a good practice because it can make reading the code confusing. The two items represented by the same name are distinct, and any change in one does not affect the other.

```
DECLARE
  v_father_name    VARCHAR2(20):='Patrick';
  v_date_of_birth  DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name    VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    ...
  
```

Which `v_date_of_birth` will be referenced in the `DBMS_OUTPUT.PUT_LINE` statement?

Variable Visibility

The **visibility** of a variable is the portion of the program where the variable can be accessed without using a qualifier. What is the visibility of each of the variables?

```
DECLARE
  v_father_name VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father's Name: ' || v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child's Name: ' || v_child_name);
  END;
  DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
END;
```

1

2



Variable Visibility continued

The `v_date_of_birth` variable declared in the outer block has scope even in the inner block. This variable is visible in the outer block. However, it is not visible in the inner block because the inner block has a local variable with the same name. The `v_father_name` variable is visible in the inner and outer blocks. The `v_child_name` variable is visible only in the inner block.

```
DECLARE
  v_father_name    VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name    VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  ...
```

What if we want to reference the outer block's `v_date_of_birth` within the inner block?



Qualifying an Identifier

A qualifier is a label given to a block. You can use this qualifier to access the variables which have scope but are not visible. In this example, the outer block has the label, <<outer>>.

```
<<outer>>
DECLARE
  v_father_name    VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name    VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  ...
```

Labeling is not limited to the outer block; you can label any block.



Qualifying an Identifier continued

Using the `outer` label to qualify the `v_date_of_birth` identifier, you can now print father's date of birth in the inner block.

```
<<outer>>
DECLARE
  v_father_name  VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20):='Mike';
    v_cate_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Father's Name: ' || v_father_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || outer.v_date_of_birth);
    DBMS_OUTPUT.PUT_LINE('Child's Name: ' || v_child_name);
    DBMS_OUTPUT.PUT_LINE('Date of Birth: ' || v_date_of_birth);
  END;
END;
```

```
Father's Name:  Patrick
Date of Birth:  20-APR-72
Child's Name:   Mike
Date of Birth:  12-DEC-02

Statement processed.
```