3

Declaring PL/SQL Variables

Objectives

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

- Recognize valid and invalid identifiers
- List the uses of variables
- Declare and initialize variables
- List and describe various data types
- Identify the benefits of using the `%TYPE` attribute
- Declare, use, and print bind variables

You have already learned about basic PL/SQL blocks and their sections. In this lesson, you learn about valid and invalid identifiers. You learn how to declare and initialize variables in the declarative section of a PL/SQL block. The lesson describes the various data types. You also learn about the `%TYPE` attribute and its benefits.

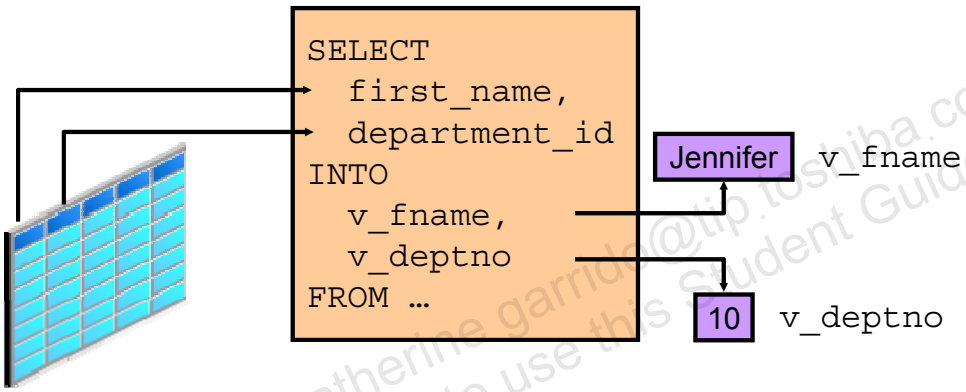
Agenda

- Introducing variables
- Examining variable data types and the `%TYPE` attribute
- Examining bind variables

Use of Variables

Variables can be used for:

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



With PL/SQL, you can declare variables, and then use them in SQL and procedural statements.

Variables are mainly used for storage of data and manipulation of stored values. Consider the PL/SQL statement in the slide. The statement retrieves `first_name` and `department_id` from the table. If you have to manipulate `first_name` or `department_id`, you have to store the retrieved value. Variables are used to temporarily store the value. You can use the value stored in these variables for processing and manipulating data. Variables can store any PL/SQL object such as variables, types, cursors, and subprograms.

Reusability is another advantage of declaring variables. After the variables are declared, you can use them repeatedly in an application by referring to them multiple times in various statements.

Requirements for Variable Names

A variable name:

- Must start with a letter
- Can include letters or numbers
- Can include special characters (such as $, _, and #)
- Must contain no more than 30 characters
- Must not include reserved words

The rules for naming a variable are listed in the slide.

Handling Variables in PL/SQL

Variables are:

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

ORACLE

You can use variables in the following ways:

- **Declare and initialize them in the declaration section:** You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the `NOT NULL` constraint on the variable. Forward references are not allowed. You must declare a variable before referencing it in other statements, including other declarative statements.
- **Use them and assign new values to them in the executable section:** In the executable section, the existing value of the variable can be replaced with a new value.
- **Pass them as parameters to PL/SQL subprograms:** Subprograms can take parameters. You can pass variables as parameters to subprograms.
- **Use them to hold the output of a PL/SQL subprogram:** Variables can be used to hold the value that is returned by a function.

Declaring and Initializing 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;
```


ORACLE

You must declare all PL/SQL identifiers in the declaration section before referencing them in the PL/SQL block. You have the option of assigning an initial value to a variable (as shown in the slide). You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, be sure that they are already declared separately in a previous statement.

In the syntax:

identifier	Is the name of the variable
CONSTANT	Constrains the variable so that its value cannot change (Constants must be initialized.)
data type	Is a scalar, composite, reference, or LOB data type (This course covers only scalar, composite, and LOB data types.)
NOT NULL	Constrains the variable so that it contains a value (NOT NULL variables must be initialized.)
expr	Is any PL/SQL expression that can be a literal expression, another variable, or an expression involving operators and functions

Note: In addition to variables, you can also declare cursors and exceptions in the declarative section. You learn about declaring cursors in the lesson titled “Using Explicit Cursors” and about exceptions in the lesson titled “Handling Exceptions.”

Declaring and Initializing PL/SQL Variables

1

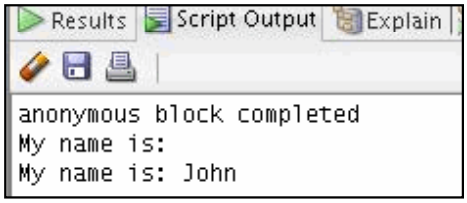
```
DECLARE
  v_myName VARCHAR2(20);
BEGIN
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
 v_myName := 'John';
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```

2

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


Examine the two code blocks in the slide.

1. In the first block, the `v_myName` variable is declared but not initialized. A value `John` is assigned to the variable in the executable section.
   - String literals must be enclosed in single quotation marks. If your string has a quotation mark as in “Today’s Date,” the string would be `'Today''s Date'`.
   - The assignment operator is: “`:=`”.
   - The `PUT_LINE` procedure is invoked by passing the `v_myName` variable. The value of the variable is concatenated with the string `'My name is:'`.
   - Output of this anonymous block is:



2. In the second block, the `v_myName` variable is declared and initialized in the declarative section. `v_myName` holds the value `John` after initialization. This value is manipulated in the executable section of the block. The output of this anonymous block is:

```
anonymous block completed
My name is: Steven
```

Delimiters in String Literals

```
DECLARE
    v_event VARCHAR2(15);
BEGIN
   v_event  := q'!Father's day!';
  DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :
   '|| v_event );
   v_event  := q'[Mother's day]';
  DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :
   '|| v_event );
END;
/
```

Resulting output

```
anonymous block completed
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day
```


If your string contains an apostrophe (identical to a single quotation mark), you must double the quotation mark, as in the following example:

```
v_event  VARCHAR2(15):='Father''s day';
```

The first quotation mark acts as the escape character. This makes your string complicated, especially if you have SQL statements as strings. You can specify any character that is not present in the string as a delimiter. The slide shows how to use the `q'` notation to specify the delimiter. The example uses `!` and `[` as delimiters. Consider the following example:

```
v_event  := q'!Father's day!';
```

You can compare this with the first example on this page. You start the string with `q'` if you want to use a delimiter. The character following the notation is the delimiter used. Enter your string after specifying the delimiter, close the delimiter, and close the notation with a single quotation mark. The following example shows how to use `[` as a delimiter:

```
v_event  := q'[Mother's day]';
```

Agenda

- Introducing variables
- Examining variable data types and the `%TYPE` attribute
- Examining bind variables

Types of Variables

- PL/SQL variables:
  - Scalar
  - Reference
  - Large object (LOB)
  - Composite
- Non-PL/SQL variables: Bind variables

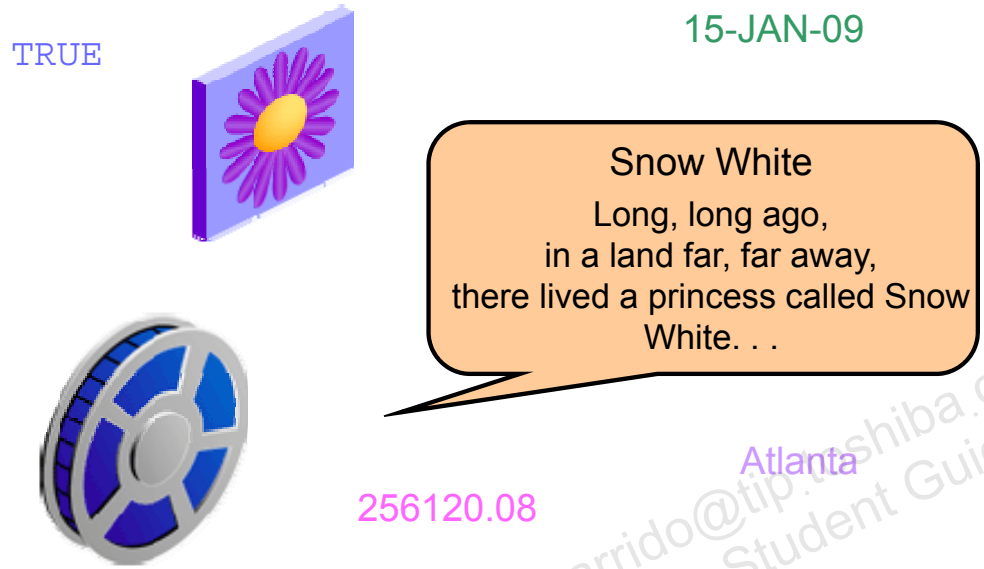
Every PL/SQL variable has a data type, which specifies a storage format, constraints, and a valid range of values. PL/SQL supports several data type categories, including scalar, reference, large object (LOB), and composite.

- **Scalar data types:** Scalar data types hold a single value. The value depends on the data type of the variable. For example, the `v_myName` variable in the example in the section "Declaring and Initializing PL/SQL Variables" (in this lesson) is of type `VARCHAR2`. Therefore, `v_myName` can hold a string value. PL/SQL also supports Boolean variables.
- **Reference data types:** Reference data types hold values, called *pointers*, which point to a storage location.
- `LOB` **data types:** `LOB` data types hold values, called *locators*, which specify the location of large objects (such as graphic images) that are stored outside the table.
- **Composite data types:** Composite data types are available by using PL/SQL *collection* and *record* variables. PL/SQL collections and records contain internal elements that you can treat as individual variables.

Non-PL/SQL variables include host language variables declared in precompiler programs, screen fields in Forms applications, and host variables. You learn about host variables later in this lesson.

For more information about `LOB`s, see the *PL/SQL User's Guide and Reference*.

Types of Variables



The slide illustrates the following data types:

- `TRUE` represents a Boolean value.
- 15-JAN-09 represents a `DATE`.
- The image represents a `BLOB`.
- The text in the callout can represent a `VARCHAR2` data type or a `CLOB`.
- 256120.08 represents a `NUMBER` data type with precision and scale.
- The film reel represents a `BFILE`.
- The city name *Atlanta* represents a `VARCHAR2` data type.

Guidelines for Declaring and Initializing PL/SQL Variables

- Follow consistent naming conventions.
- Use meaningful identifiers for variables.
- Initialize variables that are designated as `NOT NULL` and `CONSTANT`.
- Initialize variables with the assignment operator (`:=`) or the `DEFAULT` keyword:

```
v_myName VARCHAR2(20):='John';
```

```
v_myName VARCHAR2(20) DEFAULT 'John';
```

- Declare one identifier per line for better readability and code maintenance.

Here are some guidelines to follow when you declare PL/SQL variables.

- Follow consistent naming conventions—for example, you might use `name` to represent a variable and `c_name` to represent a constant. Similarly, to name a variable, you can use `v_fname`. The key is to apply your naming convention consistently for easier identification.
- Use meaningful and appropriate identifiers for variables. For example, consider using `salary` and `sal_with_commission` instead of `salary1` and `salary2`.
- If you use the `NOT NULL` constraint, you must assign a value when you declare the variable.
- In constant declarations, the `CONSTANT` keyword must precede the type specifier. The following declaration names a constant of `NUMBER` type and assigns the value of 50,000 to the constant. A constant must be initialized in its declaration; otherwise, you get a compilation error. After initializing a constant, you cannot change its value.

```
sal CONSTANT NUMBER := 50000.00;
```

Guidelines for Declaring PL/SQL Variables

- Avoid using column names as identifiers.

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

- Use the `NOT NULL` constraint when the variable must hold a value.

- Initialize the variable to an expression with the assignment operator (`:=`) or with the `DEFAULT` reserved word. If you do not assign an initial value, the new variable contains `NULL` by default until you assign a value. To assign or reassign a value to a variable, you write a PL/SQL assignment statement. However, it is good programming practice to initialize all variables.
- Two objects can have the same name only if they are defined in different blocks. Where they coexist, you can qualify them with labels and use them.
- Avoid using column names as identifiers. If PL/SQL variables occur in SQL statements and have the same name as a column, the Oracle Server assumes that it is the column that is being referenced. Although the code example in the slide works, code that is written using the same name for a database table and a variable is not easy to read or maintain.
- Impose the `NOT NULL` constraint when the variable must contain a value. You cannot assign nulls to a variable that is defined as `NOT NULL`. The `NOT NULL` constraint must be followed by an initialization clause.

```
pincode VARCHAR2(15) NOT NULL := 'Oxford';
```

Naming Conventions of PL/SQL Structures Used in This Course

PL/SQL Structure	Convention	Example
Variable	`v_variable_name`	`v_rate`
Constant	`c_constant_name`	`c_rate`
Subprogram parameter	`p_parameter_name`	`p_id`
Bind (host) variable	`b_bind_name`	`b_salary`
Cursor	`cur_cursor_name`	`cur_emp`
Record	`rec_record_name`	`rec_emp`
Type	`type_name_type`	`ename_table_type`
Exception	`e_exception_name`	`e_products_invalid`
File handle	`f_file_handle_name`	`f_file`

ORACLE

The table in the slide displays some examples of the naming conventions for PL/SQL structures that are used in this course.

Scalar Data Types

- Hold a single value
- Have no internal components

TRUE

15-JAN-09

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

256120.08

Atlanta

PL/SQL provides a variety of predefined data types. For instance, you can choose from integer, floating point, character, Boolean, date, collection, and LOB types. This lesson covers the basic types that are used frequently in PL/SQL programs.

A scalar data type holds a single value and has no internal components. Scalar data types can be classified into four categories: number, character, date, and Boolean. Character and number data types have subtypes that associate a base type to a constraint. For example, `INTEGER` and `POSITIVE` are subtypes of the `NUMBER` base type.

For more information about scalar data types (as well as a complete list), see the *PL/SQL User's Guide and Reference*.

Base Scalar Data Types

- `CHAR [(maximum_length)]`
- `VARCHAR2 (maximum_length)`
- `NUMBER [(precision, scale)]`
- `BINARY_INTEGER`
- `PLS_INTEGER`
- `BOOLEAN`
- `BINARY_FLOAT`
- `BINARY_DOUBLE`

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.
NUMBER [(*precision, scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 through 38. The scale *s* can range from –84 through 127.
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647

Data Type	Description
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 values. In Oracle Database 11*g*, the PLS_INTEGER and BINARY_INTEGER data types are identical. The arithmetic operations on PLS_INTEGER and BINARY_INTEGER values are faster than on NUMBER values.
BOOLEAN	Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, and NULL
BINARY_FLOAT	Represents floating-point number in IEEE 754 format. It requires 5 bytes to store the value.
BINARY_DOUBLE	Represents floating-point number in IEEE 754 format. It requires 9 bytes to store the value.

Base Scalar Data Types

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Data Type	Description
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 second. The syntax is TIMESTAMP[(precision)], where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.
TIMESTAMP WITH TIME ZONE	The TIMESTAMP WITH TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.

Data Type	Description
TIMESTAMP WITH LOCAL TIME ZONE	The TIMESTAMP WITH LOCAL TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH LOCAL TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–9. The default is 6. This data type differs from TIMESTAMP WITH TIME ZONE 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, the Oracle server returns the value in your local session time zone.
INTERVAL YEAR TO MONTH	You use the INTERVAL YEAR TO MONTH data type to store and manipulate intervals of years and months. The syntax is INTERVAL YEAR[(precision)] TO MONTH, where precision specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–4. The default is 2.
INTERVAL DAY TO SECOND	You use the INTERVAL DAY TO SECOND data type to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is INTERVAL DAY[(precision1)] TO SECOND[(precision2)], where precision1 and precision2 specify the number of digits in the days field and seconds field, respectively. In both cases, you cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–9. The defaults are 2 and 6, respectively.

Declaring Scalar Variables

Examples:

```
DECLARE
  v_emp_job          VARCHAR2(9);
  v_count_loop       BINARY_INTEGER := 0;
  v_dept_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;
  ...
```


The examples of variable declaration shown in the slide are defined as follows:

- `v_emp_job`: Variable to store an employee job title
- `v_count_loop`: Variable to count the iterations of a loop; initialized to `0`
- `v_dept_total_sal`: Variable to accumulate the total salary for a department; initialized to `0`
- `v_orderdate`: Variable to store the ship date of an order; initialized to one week from today
- `c_tax_rate`: Constant variable for the tax rate (which never changes throughout the PL/SQL block); set to `8.25`
- `v_valid`: Flag to indicate whether a piece of data is valid or invalid; initialized to `TRUE`

%TYPE Attribute

- Is used to declare a variable according to:
  - A database column definition
  - Another declared variable
- Is prefixed with:
  - The database table and column name
  - The name of the declared variable

PL/SQL variables are usually declared to hold and manipulate data stored in a database. When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct data type and precision. If it is not, a PL/SQL error occurs during execution. If you have to design large subprograms, this can be time consuming and error prone.

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 is derived from a table in the database. When you use the `%TYPE` attribute to declare a variable, you should prefix it with the database table and column name. If you refer to a previously declared variable, prefix the variable name of the previously declared variable to the variable being declared.

Advantages of the `%TYPE` Attribute

- You can avoid errors caused by data type mismatch or wrong precision.
- You can avoid hard coding the data type of a variable.
- You need not change the variable declaration if the column definition changes. If you have already declared some variables for a particular table without using the `%TYPE` attribute, the PL/SQL block may throw errors if the column for which the variable is 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.

Declaring Variables with the %TYPE Attribute

Syntax

```
identifier     table.column_name%TYPE;
```

Examples

```
...
  v_emp_lname      employees.last_name%TYPE;
...
```

```
...
  v_balance        NUMBER(7,2);
  v_min_balance    v_balance%TYPE := 1000;
...
```


Declare variables to store the last name of an employee. The `v_emp_lname` variable is defined to be of the same data type as the `v_last_name` column in the `employees` table. The `%TYPE` attribute provides the data type of a database column.

Declare variables to store the balance of a bank account, as well as the minimum balance, which is 1,000. The `v_min_balance` variable is defined to be of the same data type as the `v_balance` variable. The `%TYPE` attribute provides the data type of a variable.

A `NOT NULL` database column constraint does not apply to variables that are declared using `%TYPE`. Therefore, if you declare a variable using the `%TYPE` attribute that uses a database column defined as `NOT NULL`, you can assign the `NULL` value to the variable.

Declaring Boolean Variables

- Only the TRUE, FALSE, and NULL values can be assigned to a Boolean variable.
- Conditional expressions use the logical operators AND and OR, and the unary operator NOT to check the variable values.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

With PL/SQL, you can compare variables in both SQL and procedural statements. These comparisons, called Boolean expressions, consist of simple or complex expressions separated by relational operators. In a SQL statement, you can use Boolean expressions to specify the rows in a table that are affected by the statement. In a procedural statement, Boolean expressions are the basis for conditional control. NULL stands for a missing, inapplicable, or unknown value.

Examples

```
emp_sal1 := 50000;
emp_sal2 := 60000;
```

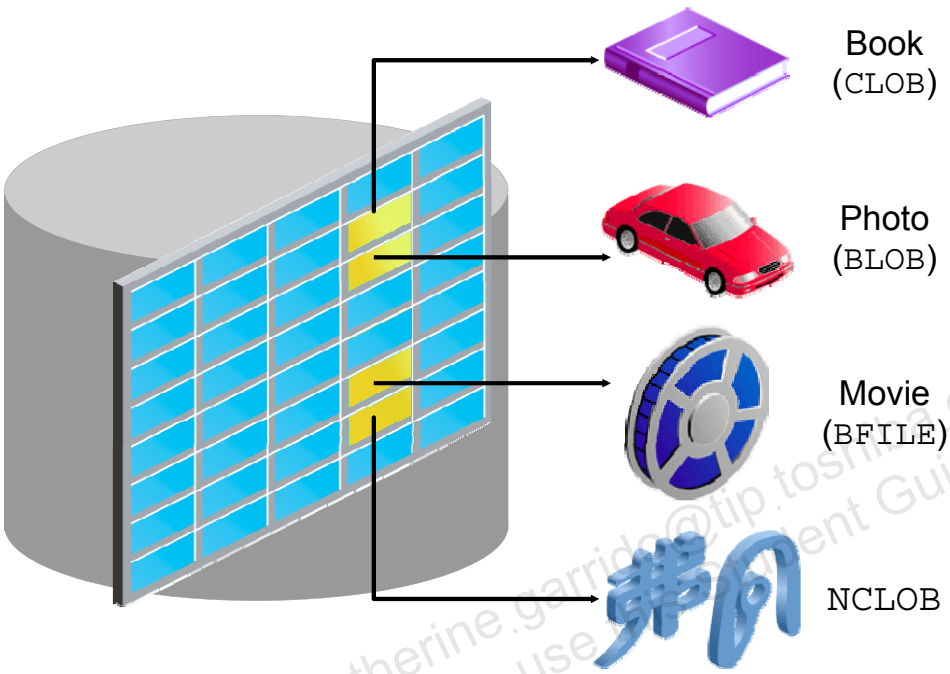
The following expression yields TRUE:

```
emp_sal1 < emp_sal2
```

Declare and initialize a Boolean variable:

```
DECLARE
 flag BOOLEAN := FALSE;
BEGIN
 flag := TRUE;
END;
/
```

LOB Data Type Variables

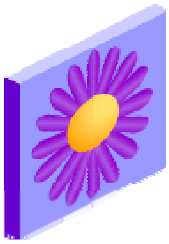


Large objects (LOBs) are meant to store a large amount of data. A database column can be of the LOB category. With the LOB category of data types (BLOB, CLOB, and so on), you can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) of up to 128 terabytes depending on the database block size. LOB data types allow efficient, random, piecewise access to data and can be attributes of an object type.

- The character large object (CLOB) data type is used to store large blocks of character data in the database.
- The binary large object (BLOB) data type is used to store large unstructured or structured binary objects in the database. When you insert or retrieve such data into or from the database, the database does not interpret the data. External applications that use this data must interpret the data.
- The binary file (BFILE) data type is used to store large binary files. Unlike other LOBs, BFILES are stored outside the database and not in the database. They could be operating system files. Only a pointer to the BFILE is stored in the database.
- The national language character large object (NCLOB) data type is used to store large blocks of single-byte or fixed-width multibyte NCHAR unicode data in the database.

Composite Data Types: Records and Collections

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	

PL/SQL Collections:

1	SMITH
2	JONES
3	NANCY
4	TIM

VARCHAR2

PLS_INTEGER

1	5000
2	2345
3	12
4	3456

NUMBER

PLS_INTEGER

As mentioned previously, a scalar data type holds a single value and has no internal components. Composite data types—called PL/SQL Records and PL/SQL Collections—have internal components that that you can treat as individual variables.

- In a PL/SQL record, the internal components can be of different data types, and are called fields. You access each field with this syntax: `record_name.field_name`. A record variable can hold a table row, or some columns from a table row. Each record field corresponds to a table column.
- In a PL/SQL collection, the internal components are always of the same data type, and are called elements. You access each element by its unique subscript. Lists and arrays are classic examples of collections. There are three types of PL/SQL collections: Associative Arrays, Nested Tables, and `VARRAY` types.

Note

- PL/SQL Records and Associative Arrays are covered in the lesson titled: “Working with Composite Data Types.”
- `NESTED TABLE` and `VARRAY` data types are covered in the course titled *Oracle Database 10g: Advanced PL/SQL* or *Oracle Database 11g: Advanced PL/SQL*.

Agenda

- Introducing variables
- Examining variable data types and the %TYPE attribute
- Examining bind variables

Bind Variables

Bind variables are:

- Created in the environment
- Also called *host* variables
- Created with the `VARIABLE` keyword*
- Used in SQL statements and PL/SQL blocks
- Accessed even after the PL/SQL block is executed
- Referenced with a preceding colon

Values can be output using the `PRINT` command.

* Required when using SQL*Plus and SQL Developer

ORACLE

Bind variables are variables that you create in a host environment. For this reason, they are sometimes called *host* variables.

Uses of Bind Variables

Bind variables are created in the environment and not in the declarative section of a PL/SQL block. Therefore, bind variables are accessible even after the block is executed. When created, bind variables can be used and manipulated by multiple subprograms. They can be used in SQL statements and PL/SQL blocks just like any other variable. These variables can be passed as run-time values into or out of PL/SQL subprograms.

Note: A bind variable is an environment variable, but is not a global variable.

Creating Bind Variables

To create a bind variable in SQL Developer, use the `VARIABLE` command. For example, you declare a variable of type `NUMBER` and `VARCHAR2` as follows:

```
VARIABLE return_code NUMBER
VARIABLE return_msg  VARCHAR2(30)
```

Viewing Values in Bind Variables

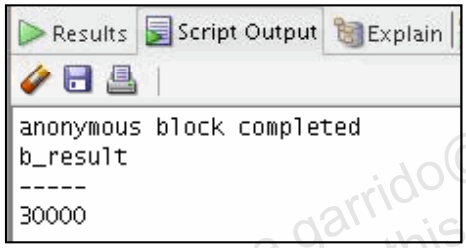
You can reference the bind variable using SQL Developer and view its value using the `PRINT` command.

Example

You can reference a bind variable in a PL/SQL program by preceding the variable with a colon.

For example, the following PL/SQL block creates and uses the bind variable `b_result`. The output resulting from the `PRINT` command is shown below the code.

```
VARIABLE b_result NUMBER
BEGIN
  SELECT (SALARY*12) + NVL(COMMISSION_PCT,0) INTO :b_result
  FROM employees WHERE employee_id = 144;
END;
/
PRINT b_result
```



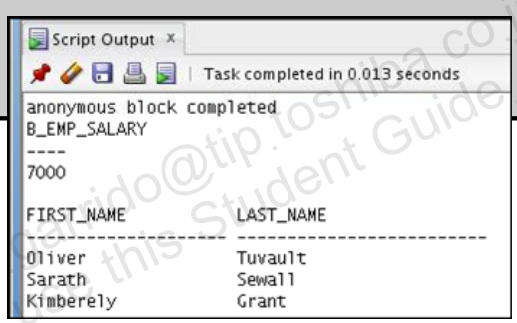
Note: If you are creating a bind variable of the `NUMBER` type, you cannot specify the precision and scale. However, you can specify the size for character strings. An Oracle `NUMBER` is stored in the same way regardless of the dimension. The Oracle Server uses the same number of bytes to store 7, 70, and .0734. It is not practical to calculate the size of the Oracle number representation from the number format, so the code always allocates the bytes needed. With character strings, the user has to specify the size so that the required number of bytes can be allocated.

Referencing Bind Variables

Example:

```
VARIABLE b_emp_salary NUMBER
BEGIN
   SELECT salary  INTO :b_emp_salary
   FROM  employees WHERE employee_id = 178;
END;
/
PRINT b_emp_salary
SELECT first_name, last_name
FROM employees
WHERE salary=:b_emp_salary;
```

Output →



As stated previously, after you create a bind variable, you can reference that variable in any other SQL statement or PL/SQL program.

In the example, `b_emp_salary` is created as a bind variable in the PL/SQL block. Then, it is used in the `SELECT` statement that follows.

When you execute the PL/SQL block shown in the slide, you see the following output:

- The `PRINT` command executes:

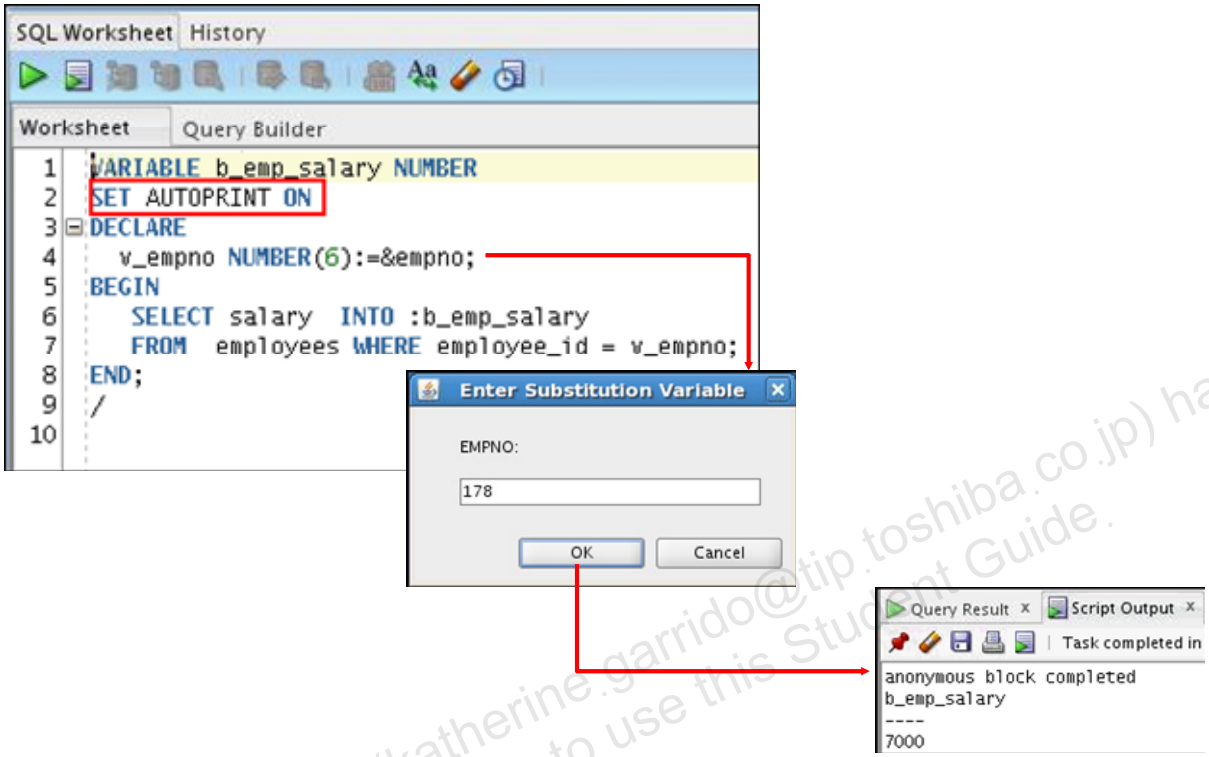
```
b_emp_salary
------------------
7000
```

- Then, the output of the SQL statement follows:

```
FIRST_NAME           LAST_NAME
------------------   -----------------------
Oliver               Tuvault
Sarath               Sewall
Kimberely            Grant
```

Note: To display all bind variables, use the `PRINT` command without a variable.

Using AUTOPRINT with Bind Variables



Use the `SET AUTOPRINT ON` command to automatically display the bind variables used in a successful PL/SQL block.

Example

In the code example:

- A bind variable named `b_emp_salary` is created and `AUTOPRINT` is turned on.
- A variable named `v_empno` is declared, and a substitution variable is used to receive user input.
- Finally, the bind variable and temporary variables are used in the executable section of the PL/SQL block.

When a valid employee number is entered—in this case 178—the output of the bind variable is automatically printed. The bind variable contains the salary for the employee number that is provided by the user.

Quiz

The `%TYPE` attribute:

a. Is used to declare a variable according to a database column definition

b. Is used to declare a variable according to a collection of columns in a database table or view

c. Is used to declare a variable according to the definition of another declared variable

d. Is prefixed with the database table and column name or the name of the declared variable

Answer: a, c, d

The `%TYPE` Attribute

PL/SQL variables are usually declared to hold and manipulate data stored in a database. When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct data type and precision. If it is not, a PL/SQL error occurs during execution. If you have to design large subprograms, this can be time consuming and error prone.

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 is derived from a table in the database. When you use the `%TYPE` attribute to declare a variable, you should prefix it with the database table and column name. If you refer to a previously declared variable, prefix the variable name of the previously declared variable to the variable being declared. The benefit of `%TYPE` is that you do not have to change the variable if the column is altered. Also, if the variable is used in any calculations, you need not worry about its precision.

The `%ROWTYPE` Attribute

The `%ROWTYPE` attribute is used to declare a record that can hold an entire row of a table or view. You learn about this attribute in the lesson titled “Working with Composite Data Types.”

Summary

In this lesson, you should have learned how to:

- Recognize valid and invalid identifiers
- Declare variables in the declarative section of a PL/SQL block
- Initialize variables and use them in the executable section
- Differentiate between scalar and composite data types
- Use the `%TYPE` attribute
- Use bind variables

ORACLE

An anonymous PL/SQL block is a basic, unnamed unit of a PL/SQL program. It consists of a set of SQL or PL/SQL statements to perform a logical function. The declarative part is the first part of a PL/SQL block and is used for declaring objects such as variables, constants, cursors, and definitions of error situations called *exceptions*.

In this lesson, you learned how to declare variables in the declarative section. You saw some of the guidelines for declaring variables. You learned how to initialize variables when you declare them.

The executable part of a PL/SQL block is the mandatory part and contains SQL and PL/SQL statements for querying and manipulating data. You learned how to initialize variables in the executable section and also how to use them and manipulate the values of variables.

Practice 3: Overview

This practice covers the following topics:

- Determining valid identifiers
- Determining valid variable declarations
- Declaring variables within an anonymous block
- Using the `%TYPE` attribute to declare variables
- Declaring and printing a bind variable
- Executing a PL/SQL block

Exercises 1, 2, and 3 are paper based.