2

Declaring PL/SQL Variables

ORACLE

Objectives

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

- **Identify 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**

ORACLE

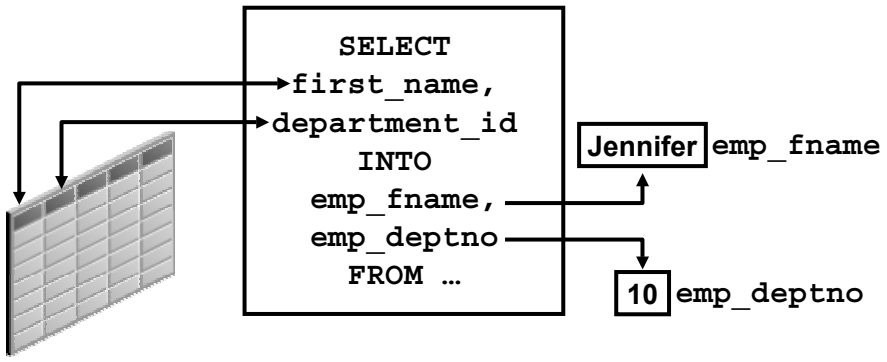
Lesson Aim

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.

Use of Variables

Variables can be used for:

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



ORACLE

Use of Variables

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 SQL statement shown in the slide. The statement retrieves the `first_name` and `department_id` from the table. If you have to manipulate the `first_name` or the `department_id`, then 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 the data. Variables can store any PL/SQL object, such as variables, types, cursors, and subprograms.

Reusability is another advantage of declaring variables. After they are declared, variables can be used repeatedly in an application by referring to them in the statements.

Identifiers

Identifiers are used for:

- **Naming a variable**
- **Providing conventions for variable names**
  - **Must start with a letter**
  - **Can include letters or numbers**
  - **Can include special characters (such as dollar sign, underscore, and pound sign)**
  - **Must limit the length to 30 characters**
  - **Must not be reserved words**

ORACLE

Identifiers

Identifiers are mainly used to provide conventions for naming variables. The rules for naming a variable are listed in the slide.

What Is the Difference Between a Variable and an Identifier?

Identifiers are names of variables. Variables are storage locations of data. Data is stored in memory. Variables point to this memory location where data can be read and modified. Identifiers are used to *name* PL/SQL objects (such as variables, types, cursors, and subprograms). Variables are used to *store* PL/SQL objects.

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**
- **Used to hold the output of a PL/SQL subprogram**

ORACLE

Handling Variables in PL/SQL

Declared and Initialized 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.

Used and Assigned New Values in the Executable Section

In the executable section, the existing value of the variable can be replaced with the new value.

Passed as Parameters to PL/SQL Subprograms

Subprograms can take parameters. You can pass variables as parameters to subprograms.

Used to Hold the Output of a PL/SQL Subprogram

You have learned that the only difference between procedures and functions is that functions must return a value. 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
  emp_hiredate    DATE;
  emp_deptno      NUMBER(2) NOT NULL := 10;
  location        VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

ORACLE

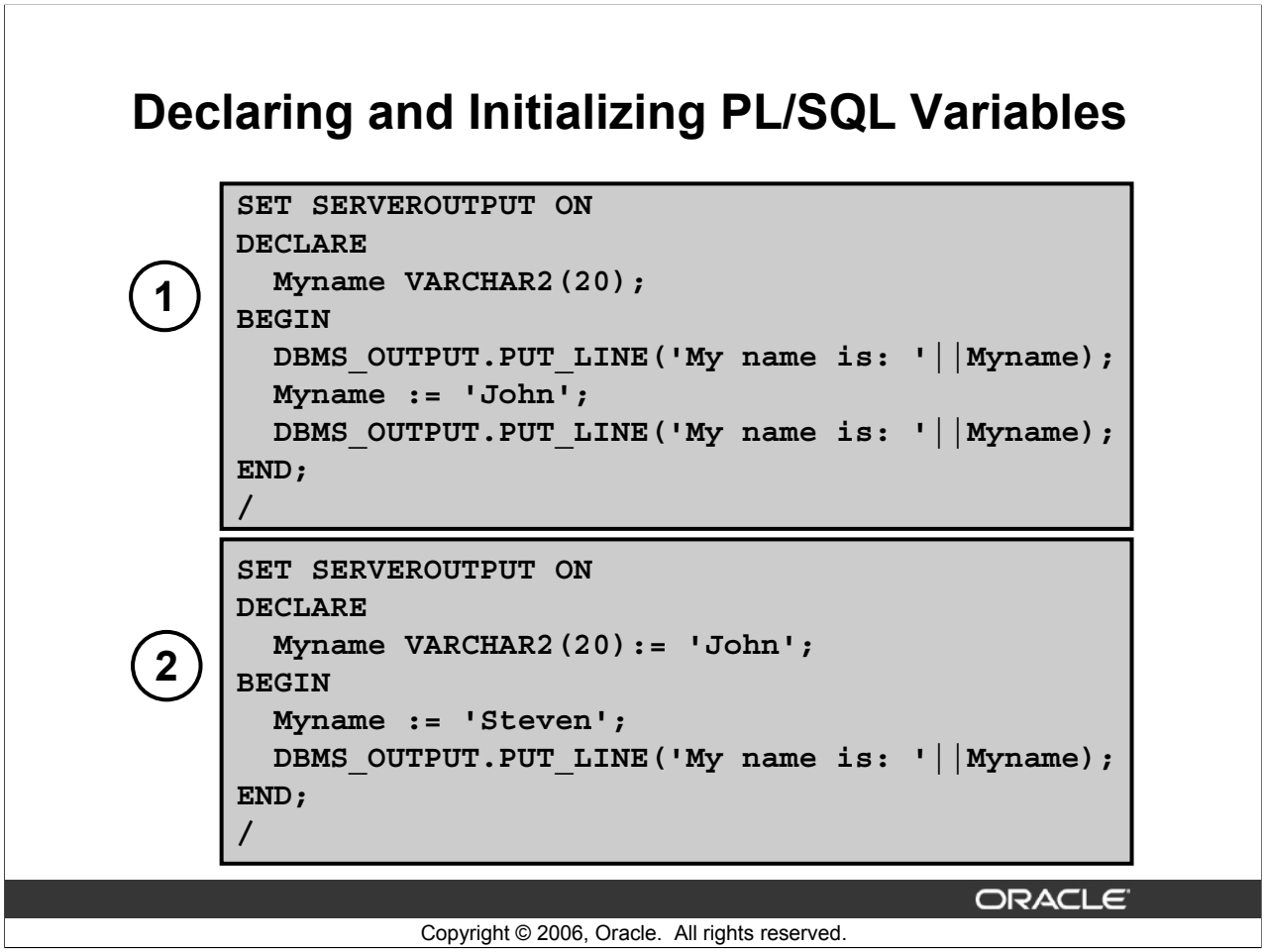
Declaring and Initializing PL/SQL Variables

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, you must be sure to declare them 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 must contain 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 how to declare cursors and exceptions later in the course.



Declaring and Initializing PL/SQL Variables (continued)

Examine the two code blocks in the slide.

1. The variable `Myname` is declared in the declarative section of the block. This variable can be accessed in the executable section of the same block. 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", then the string would be " `Today's Date`". ':=' is the assignment operator. The procedure `PUT_LINE` is invoked by passing the variable `Myname`. The value of the variable is concatenated with the string 'My name is: '. The output of this anonymous block is:

   ```
   My name is:
   My name is: John
   PL/SQL procedure successfully completed.
   ```

2. In the second block, the variable `Myname` is declared and initialized in the declarative section. `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:

   ```
   My name is: Steven
   PL/SQL procedure successfully completed.
   ```

Delimiters in String Literals

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

```
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day
PL/SQL procedure successfully completed.
```

ORACLE

Delimiters in String Literals

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

```
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 delimiter. The slide shows how to use the `q'` notation to specify the delimiter. The examples use '!' and '[' as delimiters. Consider the following example:

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

You can compare this with the first example on this notes 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:

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

Types of Variables

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

ORACLE

Types of Variables

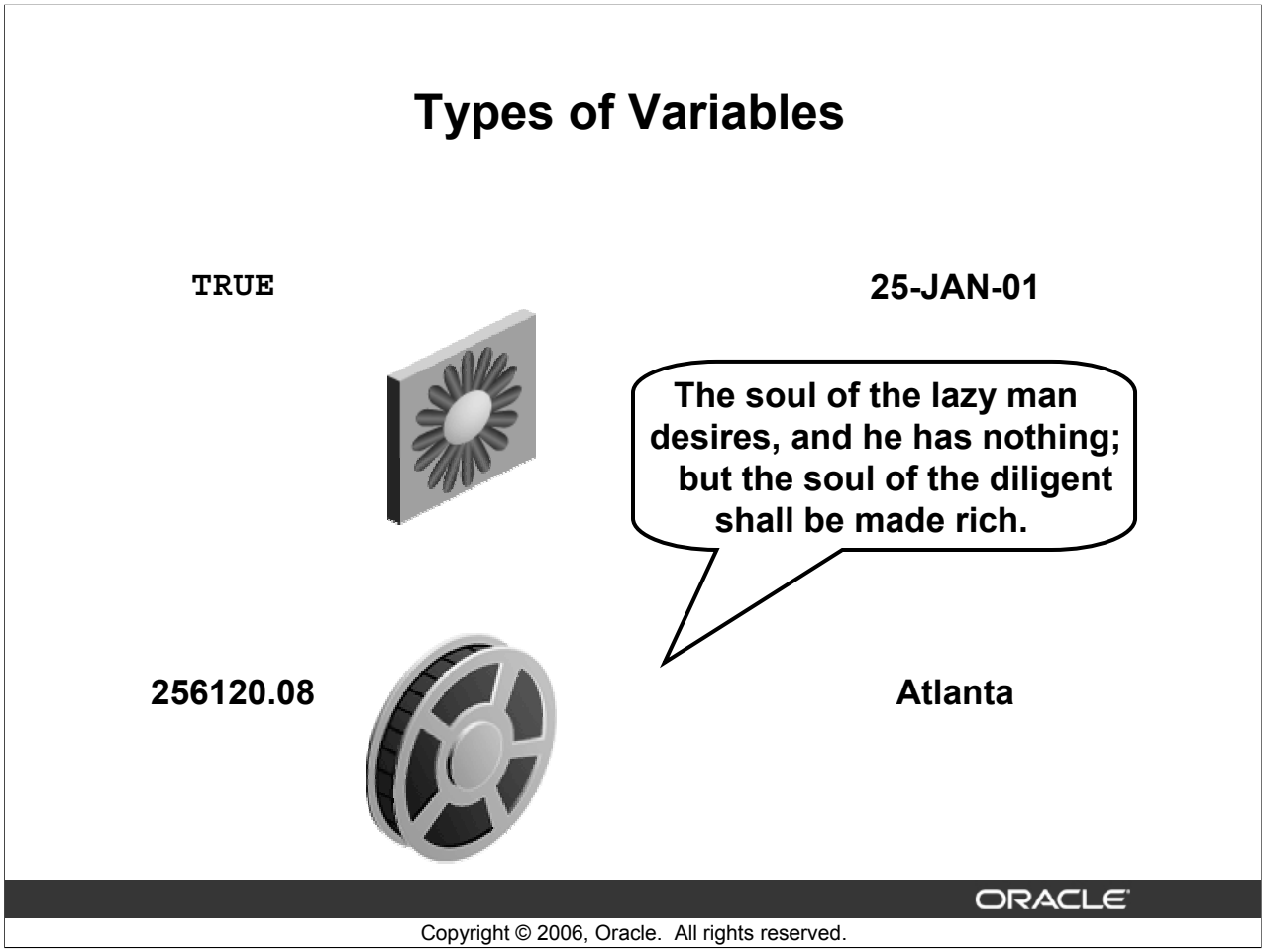
All PL/SQL variables have a data type, which specifies a storage format, constraints, and a valid range of values. PL/SQL supports five data type categories—scalar, composite, reference, large object (LOB), and object—that you can use for declaring variables, constants, and pointers.

- **Scalar data types:** Scalar data types hold a single value. The value depends on the data type of the variable. For example, the variable `Myname` in the example in slide 7 is of type `VARCHAR2`. Therefore, `Myname` can hold a string value. PL/SQL also supports Boolean variables.
- **Composite data types:** Composite data types contain internal elements that are either scalar or composite. Record and table are examples of composite data types.
- **Reference data types:** Reference data types hold values, called *pointers*, that point to a storage location.
- **LOB data types:** LOB data types hold values, called *locators*, that specify the location of large objects (such as graphic images) that are stored out of line.

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

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

Types of Variables



Types of Variables (continued)

The slide illustrates the following data types:

- `TRUE` represents a Boolean value.
- 25-JAN-01 represents a `DATE`.
- The image represents a `BLOB`.
- The text of the proverb 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`.

Guidelines for Declaring and Initializing PL/SQL Variables

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

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

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

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

ORACLE

Guidelines for Declaring and Initializing PL/SQL Variables

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

- Follow naming conventions: for example, `name` to represent a variable and `c_name` to represent a constant.
- Use meaningful and appropriate names 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 keyword `CONSTANT` must precede the type specifier. The following declaration names a constant of `NUMBER` subtype `REAL` 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 REAL := 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.**

ORACLE

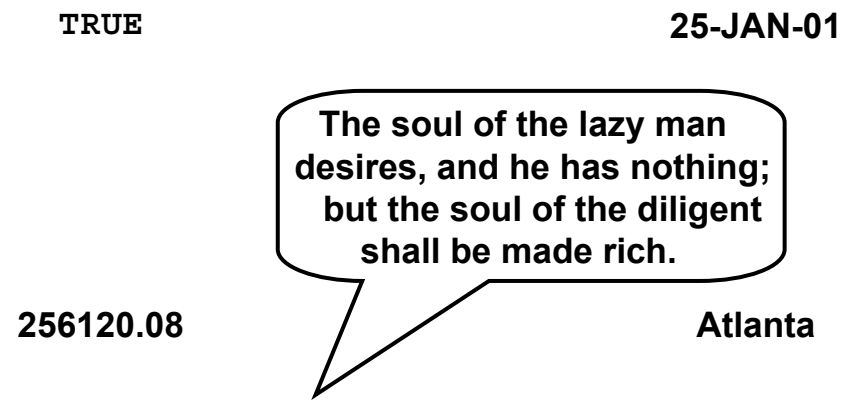
Guidelines for Declaring PL/SQL Variables

- 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. 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 example code in the slide works, code that is written using the same name for a database table and variable name 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 defined as NOT NULL. The NOT NULL constraint must be followed by an initialization clause.

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

Scalar Data Types

- **Hold a single value**
- **Have no internal components**



ORACLE

Scalar Data Types

Every constant, variable, and parameter has a data type that specifies a storage format, constraints, and valid range of values. 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 chapter 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 and the complete list of scalar data types, refer to the *PL/SQL User's Guide and Reference*.

Base Scalar Data Types

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

ORACLE

Base Scalar Data Types

Data Type	Description
CHAR [(*maximum_length*)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1.
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(*precision, scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from –84 to 127.
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647.

Base Scalar Data Types (continued)

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 10*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	New data type introduced in Oracle Database 10*g*. Represents floating-point number in IEEE 754 format. Requires 5 bytes to store the value.
BINARY_DOUBLE	New data type introduced in Oracle Database 10*g*. Represents floating-point number in IEEE 754 format. 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**

ORACLE

Base Scalar Data Types (continued)

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 9999 A.D.
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. 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.
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. 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.

Base Scalar Data Types (continued)

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.

BINARY_FLOAT and BINARY_DOUBLE

- Represent floating point numbers in IEEE 754 format
- Offer better interoperability and operational speed
- Store values beyond the values that the data type NUMBER can store
- Provide the benefits of closed arithmetic operations and transparent rounding

ORACLE

BINARY_FLOAT and BINARY_DOUBLE

BINARY_FLOAT and BINARY_DOUBLE are new data types introduced in Oracle database 10*g*.

- **Represent floating point numbers in IEEE 754 format:** You can use these data types for scientific calculations and also for data exchange between programs that follow the IEEE (Institute of Electrical and Electronics Engineers) format.
- **Benefits:** Many computer systems support IEEE 754 floating-point operations through native processor instructions. These types are efficient for intensive computations involving floating-point data. Interaction with such programs is made easier because Oracle supports the same format to which these two data types adhere.
- **Better interoperability and operational speed:** Interoperability is mainly due to the format of these two data types. These data types improve performance in number-crunching operations such as processing scientific data.
- **Store values beyond Oracle NUMBER:** BINARY_FLOAT requires 5 bytes and BINARY_DOUBLE requires 9 bytes as opposed to Oracle NUMBER, which uses anywhere between 1 and 22 bytes. These data types meet the demand for a numeric data type that can store numeric data beyond the range of NUMBER.

BINARY_FLOAT and BINARY_DOUBLE (continued)

- **Closed arithmetic operations and transparent rounding:** All arithmetic operations with BINARY_FLOAT and BINARY_DOUBLE are closed; that is, an arithmetic operation produces a normal or special value. You need not worry about explicit conversion. For example, multiplying a BINARY_FLOAT number with another BINARY_FLOAT results in a BINARY_FLOAT number. Dividing a BINARY_FLOAT by zero is undefined and actually results in the special value Inf (Infinite). Operations on these data types are subject to rounding, which is transparent to PL/SQL users. The default mode is rounding to the nearest binary place. Most financial applications require decimal rounding behavior, whereas purely scientific applications may not.

  Example

```
SET SERVEROUTPUT ON
DECLARE
  bf_var BINARY_FLOAT;
  bd_var BINARY_DOUBLE;
BEGIN
  bf_var := 270/35f;
  bd_var := 140d/0.35;
  DBMS_OUTPUT.PUT_LINE('bf: '|| bf_var);
  DBMS_OUTPUT.PUT_LINE('bd: '|| bd_var);
END;
/
bf: 7.71428585E+000
bd: 4.0E+002
PL/SQL procedure successfully completed.
```

Declaring Scalar Variables

Examples

```
DECLARE
  emp_job         VARCHAR2(9);
  count_loop      BINARY_INTEGER := 0;
  dept_total_sal  NUMBER(9,2) := 0;
  orderdate       DATE := SYSDATE + 7;
  c_tax_rate      CONSTANT NUMBER(3,2) := 8.25;
  valid           BOOLEAN NOT NULL := TRUE;
  ...
```

ORACLE

Declaring Scalar Variables

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

- **emp_job:** Variable to store an employee job title
- **count_loop:** Variable to count the iterations of a loop; initialized to `0`
- **dept_total_sal:** Variable to accumulate the total salary for a department; initialized to `0`
- **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`
- **valid:** Flag to indicate whether a piece of data is valid or invalid; initialized to `TRUE`

%TYPE Attribute

The %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**
  - **The name of the declared variable**

ORACLE

%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 to the attribute.

%TYPE Attribute (continued)

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

```
...
  emp_lname      employees.last_name%TYPE;
  balance        NUMBER(7,2);
  min_balance    balance%TYPE := 1000;
...
```

ORACLE

Declaring Variables with the %TYPE Attribute

Declare variables to store the last name of an employee. The variable `emp_lname` is defined to be of the same data type as the `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 variable `min_balance` is defined to be of the same data type as the variable `balance`. 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 values `TRUE`, `FALSE`, and `NULL` 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.**

ORACLE

Declaring Boolean Variables

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;
/
```

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**

ORACLE

Bind Variables

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. Variables declared in a PL/SQL block are available only when you execute the block. After the block is executed, the memory used by the variable is freed. However, bind variables are accessible even after the block is executed. When created, therefore, 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.

Creating Bind Variables

To create a bind variable in *i*SQL*Plus or in SQL*Plus, 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)
```

Both SQL*Plus and *i*SQL*Plus can reference the bind variable, and *i*SQL*Plus can display its value through the SQL*Plus `PRINT` command.

Bind Variables (continued)

Example

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

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

RESULT
30000

Note: If you are creating a bind variable of type `NUMBER`, 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 size is required from the user so that the required number of bytes can be allocated.

Printing Bind Variables from the Environment

To display the current value of bind variables in the *i*SQL*Plus environment, use the `PRINT` command. However, `PRINT` cannot be used inside a PL/SQL block because it is an *i*SQL*Plus command. Note how the variable `result` is printed using the `PRINT` command in the code block shown above.

Printing Bind Variables

Example

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

ORACLE

Printing Bind Variables

In *i*SQL*Plus, you can display the value of a bind variable by using the `PRINT` command. When you execute the PL/SQL block shown in the slide, you see the following output when the `PRINT` command executes.

EMP_SALARY
7000

`emp_salary` is a bind variable. You can now use this variable in any SQL statement or PL/SQL program. Note the SQL statement that uses the bind variable. The output of the SQL statement is:

FIRST_NAME	LAST_NAME
Oliver	Tuvault
Sarath	Sewall
Kimberely	Grant

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

Printing Bind Variables

Example

```
VARIABLE emp_salary NUMBER
SET AUTOPRINT ON
BEGIN
   SELECT salary  INTO :emp_salary
   FROM  employees WHERE employee_id = 178;
END;
/
```

ORACLE

Printing Bind Variables (continued)

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

Substitution Variables

- Are used to get user input at run time
- Are referenced within a PL/SQL block with a preceding ampersand
- Are used to avoid hard-coding values that can be obtained at run time

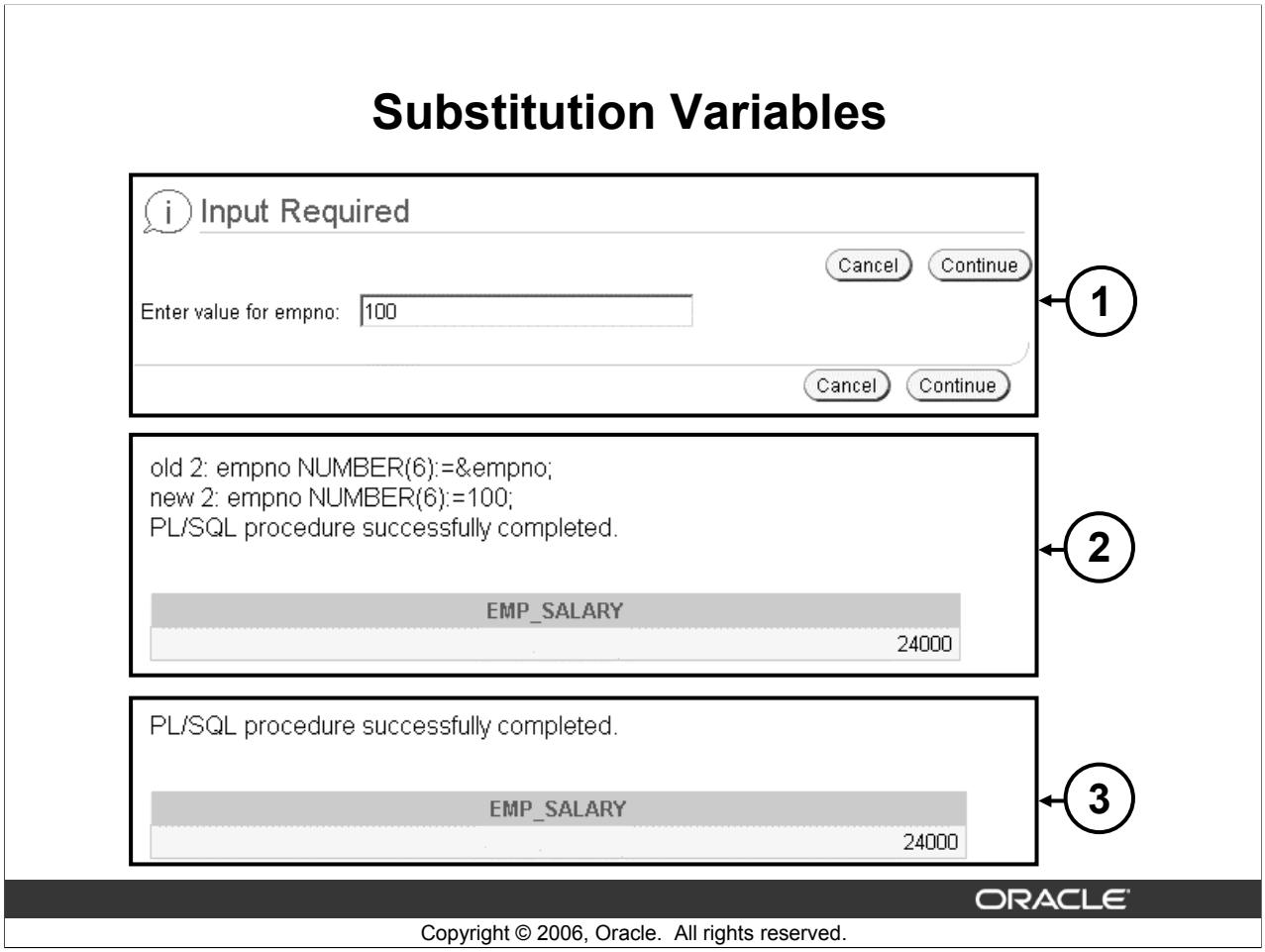
```
VARIABLE emp_salary NUMBER
SET AUTOPRINT ON
DECLARE
  empno NUMBER(6):=&empno;
BEGIN
  SELECT salary  INTO :emp_salary
  FROM  employees WHERE employee_id = empno;
END;
/
```

ORACLE

Substitution Variables

In the *i*SQL*Plus environment, *i*SQL*Plus substitution variables can be used to pass run-time values into a PL/SQL block. You can reference substitution variables in SQL statements (and within a PL/SQL block) with a preceding ampersand. The text values are substituted into the PL/SQL block before the PL/SQL block is executed. Therefore, you cannot substitute different values for the substitution variables by using a loop. Even if you include the variable in a loop, you are prompted only once to enter the value. Only one value will replace the substitution variable.

When you execute the block in the slide, *i*SQL*Plus prompts you to enter a value for `empno`, which is the substitution variable.

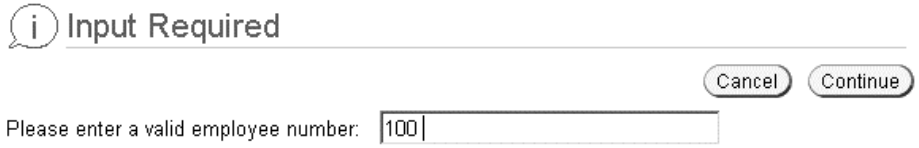


Substitution Variables (continued)

1. When you execute the block in the previous slide, *i*SQL*Plus prompts you to enter a value for `empno`, which is the substitution variable. By default, the prompt message is "Enter value for <*substitution variable*>." Enter a value as shown in the slide and click the Continue button.
2. You see the output shown in the slide. Note that *i*SQL*Plus prints both the old value and the new value for the substitution variable. You can disable this behavior by using the `SET VERIFY OFF` command.
3. This is the output after using the `SET VERIFY OFF` command.

Prompt for Substitution Variables

```
SET VERIFY OFF
VARIABLE emp_salary NUMBER
ACCEPT empno PROMPT 'Please enter a valid employee
number: '
SET AUTOPRINT ON
DECLARE
  empno NUMBER(6):= &empno;
BEGIN
  SELECT salary  INTO :emp_salary  FROM  employees
  WHERE employee_id = empno;
END;
/
```



ORACLE

Prompt for Substitution Variables

The default prompt message in the preceding slide was "Enter value for <*substitution variable*>."

Use the `PROMPT` command to change the message (as shown in this slide). This is an *i*SQL*Plus command and therefore cannot be included in the PL/SQL block.

Using DEFINE for a User Variable

Example

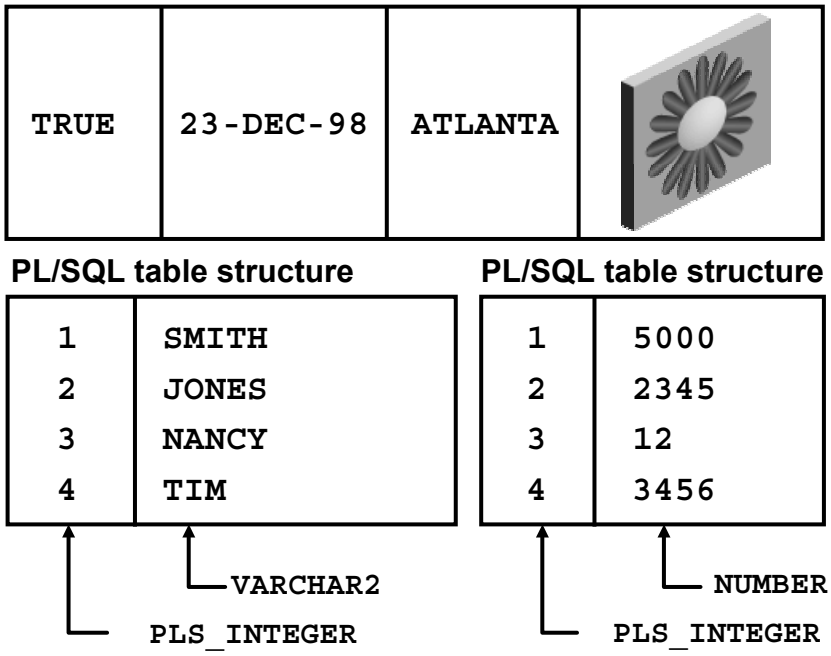
```
SET VERIFY OFF
DEFINE lname= Urman
DECLARE
  fname VARCHAR2(25);
BEGIN
  SELECT first_name INTO fname FROM employees
  WHERE last_name='&lname';
END;
/
```

ORACLE

Using DEFINE for a User Variable

The DEFINE command specifies a user variable and assigns it a CHAR value. You can define variables of CHAR data type only. Even though you enter the number 50000, *i*SQL*Plus assigns a CHAR value to a variable consisting of the characters 5,0,0,0, and 0. You can reference such variables with a preceding ampersand (&), as shown in the slide.

Composite Data Types



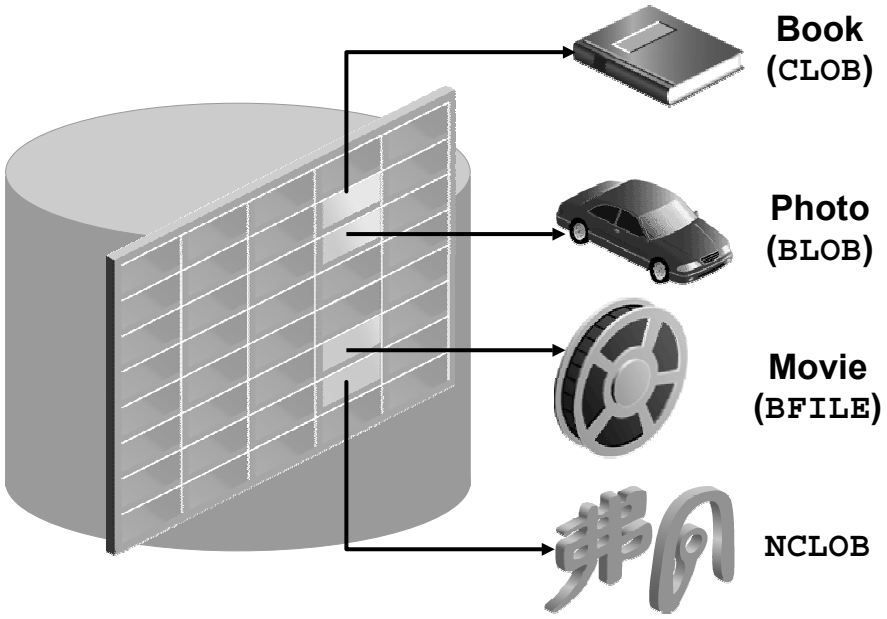
ORACLE

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 `TABLE`, `RECORD`, `NESTED TABLE`, and `VARRAY` types.

Use the `TABLE` data type to reference and manipulate collections of data as a whole object. Use the `RECORD` data type to treat related but dissimilar data as a logical unit. `NESTED TABLE` and `VARRAY` data types are covered in the *Oracle Database 10g: Develop PL/SQL Program Units* course.

LOB Data Type Variables



ORACLE

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) up to 4 GB in size. LOB data types allow efficient, random, piecewise access to the 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 to and 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 not stored in the database. BFILEs are stored outside 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.

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

Summary

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 utilize them and manipulate the values of variables.

Practice 2: 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**

ORACLE

Practice 2: Overview

Exercises 1, 2, and 3 are paper based.

Practice 2

Note: It is recommended to use *i*SQL*Plus for this practice.

1. Identify valid and invalid identifier names:
   a. `today`
   b. `last_name`
   c. `today's_date`
   d. `Number_of_days_in_February_this_year`
   e. `Isleap$year`
   f. `#number`
   g. `NUMBER#`
   h. `number1to7`

2. Identify valid and invalid variable declaration and initialization:
   a. `number_of_copies PLS_INTEGER;`
   b. `printer_name constant VARCHAR2(10);`
   c. `deliver_to VARCHAR2(10):=Johnson;`
   d. `by_when DATE:= SYSDATE+1;`

3. Examine the following anonymous block and choose the appropriate statement.

```
SET SERVEROUTPUT ON
DECLARE
  fname VARCHAR2(20);
  lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
  DBMS_OUTPUT.PUT_LINE( FNAME ||' ' ||lname);
END;
/
```

   a. The block will execute successfully and print 'fernandez'
   b. The block will give an error because the `fname` variable is used without initializing.
   c. The block will execute successfully and print 'null fernandez'
   d. The block will give an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.
   e. The block will give an error because the variable `FNAME` is not declared.

4. Create an anonymous block. In *i*SQL*Plus, load the script `lab_01_02_soln.sql`, which you created in question 2 of practice 1.
   a. Add a declarative section to this PL/SQL block. In the declarative section, declare the following variables:
      1. Variable `today` of type `DATE`. Initialize `today` with `SYSDATE`.
      2. Variable `tomorrow` of type `today`. Use `%TYPE` attribute to declare this variable.
   b. In the executable section initialize the variable `tomorrow` with an expression, which calculates tomorrow's date (add one to the value in `today`). Print the value of `today` and `tomorrow` after printing 'Hello World'

Practice 2 (continued)

c. Execute and save this script as `lab_02_04_soln.sql`. Sample output is shown below.

```
Hello World
TODAY IS : 12-JAN-04
TOMORROW IS : 13-JAN-04
PL/SQL procedure successfully completed.
```

5. Edit the `lab_02_04_soln.sql` script.
   a. Add code to create two bind variables.
      Create bind variables `basic_percent` and `pf_percent` of type `NUMBER`.
   b. In the executable section of the PL/SQL block assign the values 45 and 12 to `basic_percent` and `pf_percent` respectively.
   c. Terminate the PL/SQL block with "/" and display the value of the bind variables by using the `PRINT` command.
   d. Execute and save your script file as `lab_02_05_soln.sql`. Sample output is shown below.

```
Hello World
TODAY IS : 12-JAN-04
TOMORROW IS : 13-JAN-04
PL/SQL procedure successfully completed.
```

BASIC_PERCENT
45

Next Page

Click the Next Page button.

PF_PERCENT
12