1 Declaring Variables

Schedule:	Timing	Topic
	60 minutes	Lecture
	30 minutes	Practice
	90 minutes	Total

Objectives

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

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

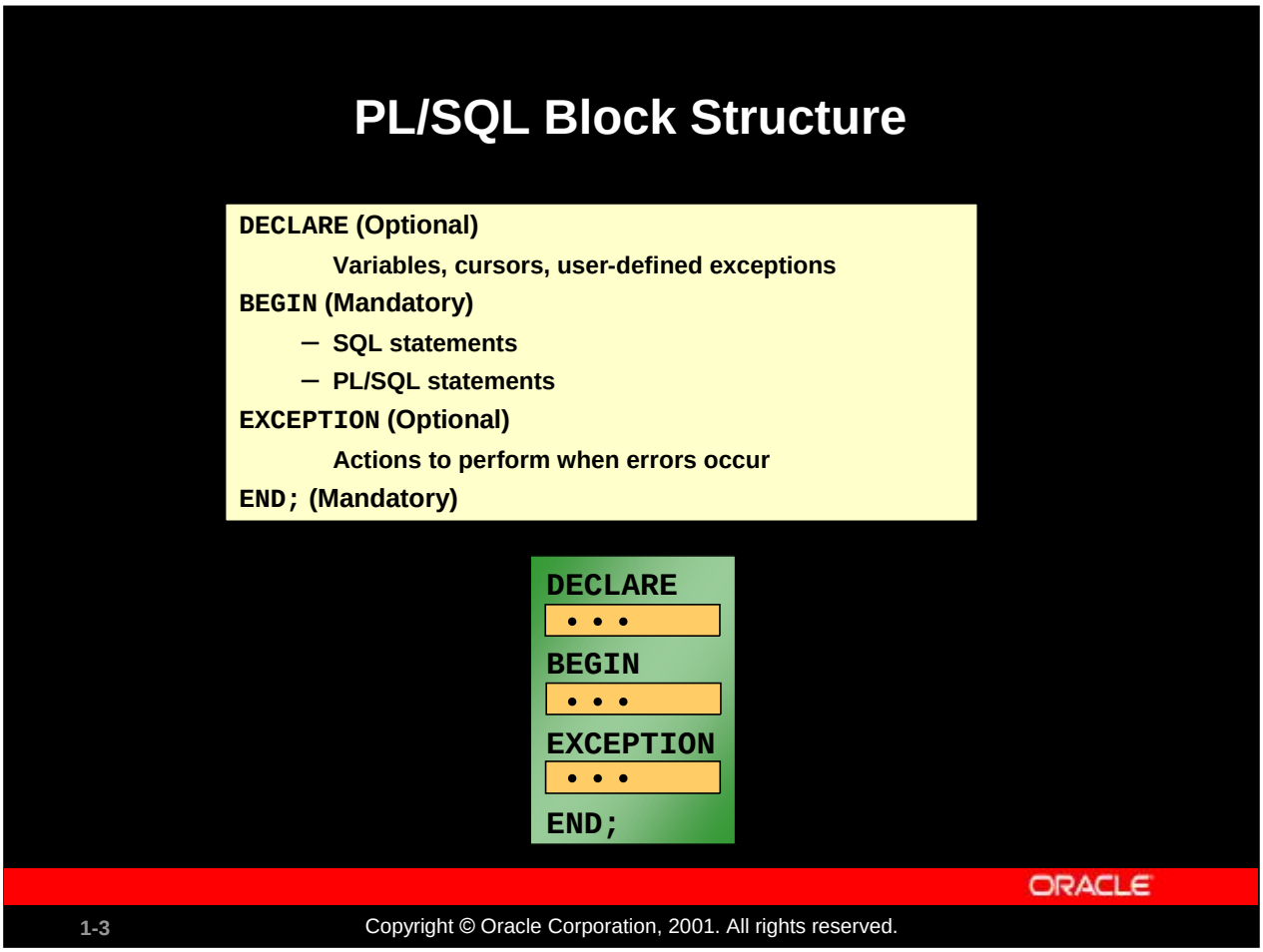
ORACLE

Lesson Aim

This lesson presents the basic rules and structure for writing and executing PL/SQL blocks of code. It also shows you how to declare variables and assign data types to them.

Instructor Note

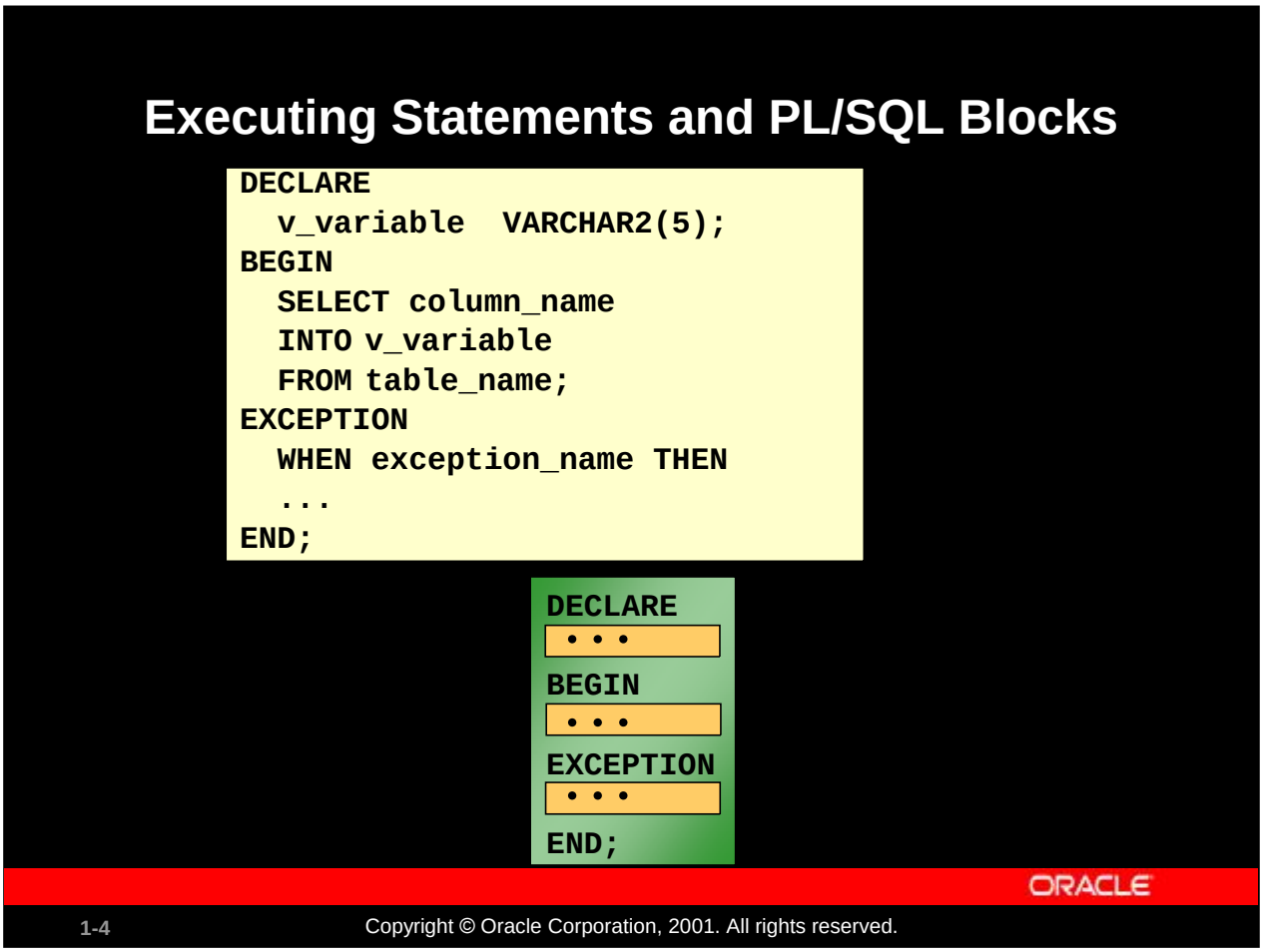
Inform the class that *i*SQL*Plus is used throughout this course for the execution of demonstrations and lesson practices.



PL/SQL Block Structure

PL/SQL is a block-structured language, meaning that programs can be divided into logical blocks. A PL/SQL block consists of up to three sections: declarative (optional), executable (required), and exception handling (optional). The following table describes the three sections:

Section	Description	Inclusion
Declarative	Contains all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and declarative sections	Optional
Executable	Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block	Mandatory
Exception handling	Specifies the actions to perform when errors and abnormal conditions arise in the executable section	Optional



Executing Statements and PL/SQL Blocks

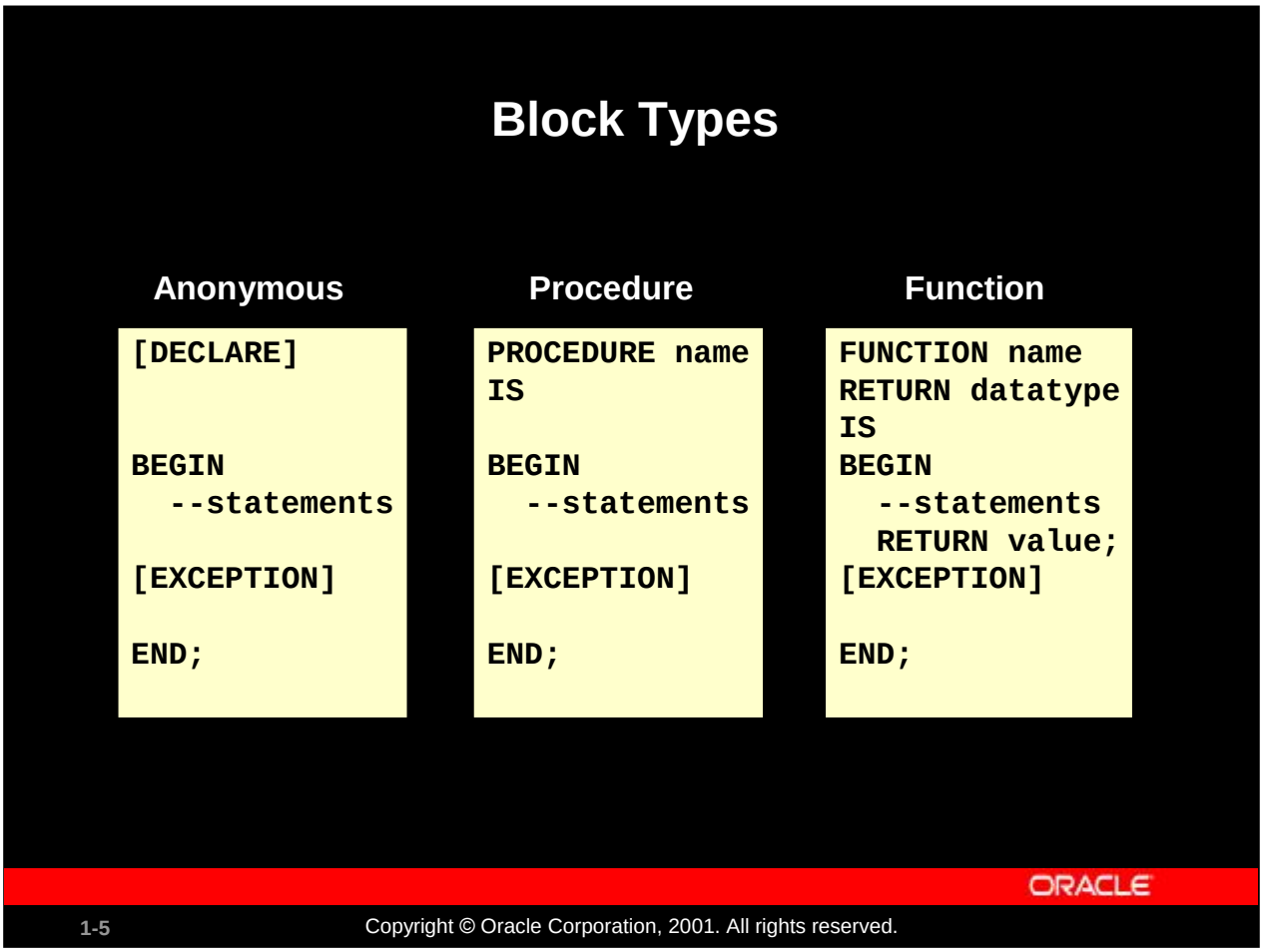
- Place a semicolon (;) at the end of a SQL statement or PL/SQL control statement.
- When the block is executed successfully, without unhandled errors or compile errors, the message output should be as follows:

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

- Section keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not followed by semicolons.
- `END` and all other PL/SQL statements require a semicolon to terminate the statement.
- You can string statements together on the same line, but this method is not recommended for clarity or editing.

Note: In PL/SQL, an error is called an exception.

With modularity you can break an application down into manageable, well-defined modules. Through successive refinement, you can reduce a complex problem to a set of simple problems that have easy-to-implement solutions. PL/SQL meets this need with program units, which include blocks, subprograms, and packages.



Block Types

A PL/SQL program comprises one or more blocks. These blocks can be entirely separate or nested one within another. The basic units (procedures and functions, also known as subprograms, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested subblocks. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Anonymous Blocks

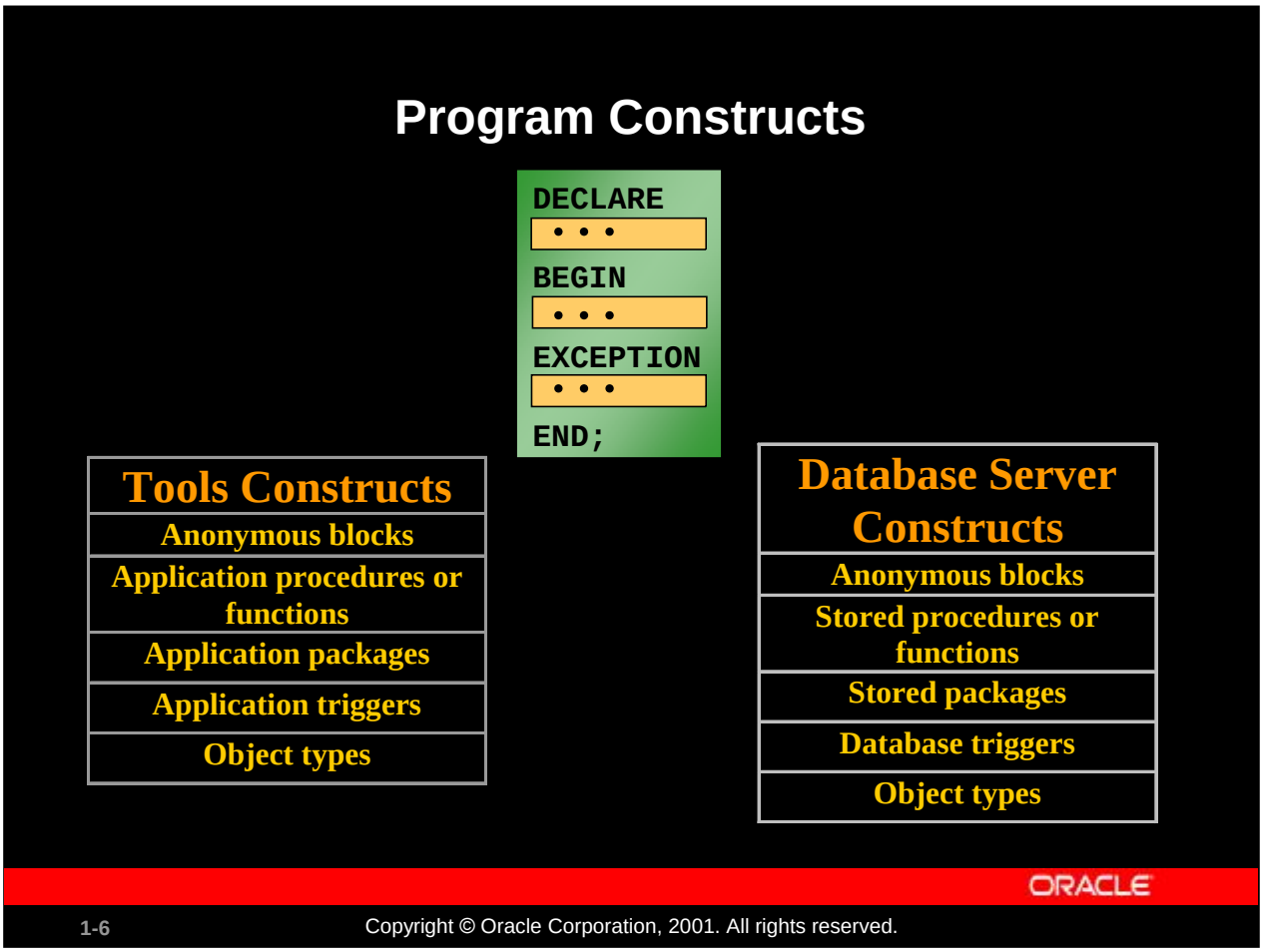
Anonymous blocks are unnamed blocks. They are declared at the point in an application where they are to be executed and are passed to the PL/SQL engine for execution at run time. You can embed an anonymous block within a precompiler program and within *i*SQL*Plus or Server Manager. Triggers in Oracle Developer components consist of such blocks.

Subprograms

Subprograms are named PL/SQL blocks that can accept parameters and can be invoked. You can declare them either as procedures or as functions. Generally use a procedure to perform an action and a function to compute a value.

You can store subprograms at the server or application level. Using Oracle Developer components (Forms, Reports, and Graphics), you can declare procedures and functions as part of the application (a form or report) and call them from other procedures, functions, and triggers (see next page) within the same application whenever necessary.

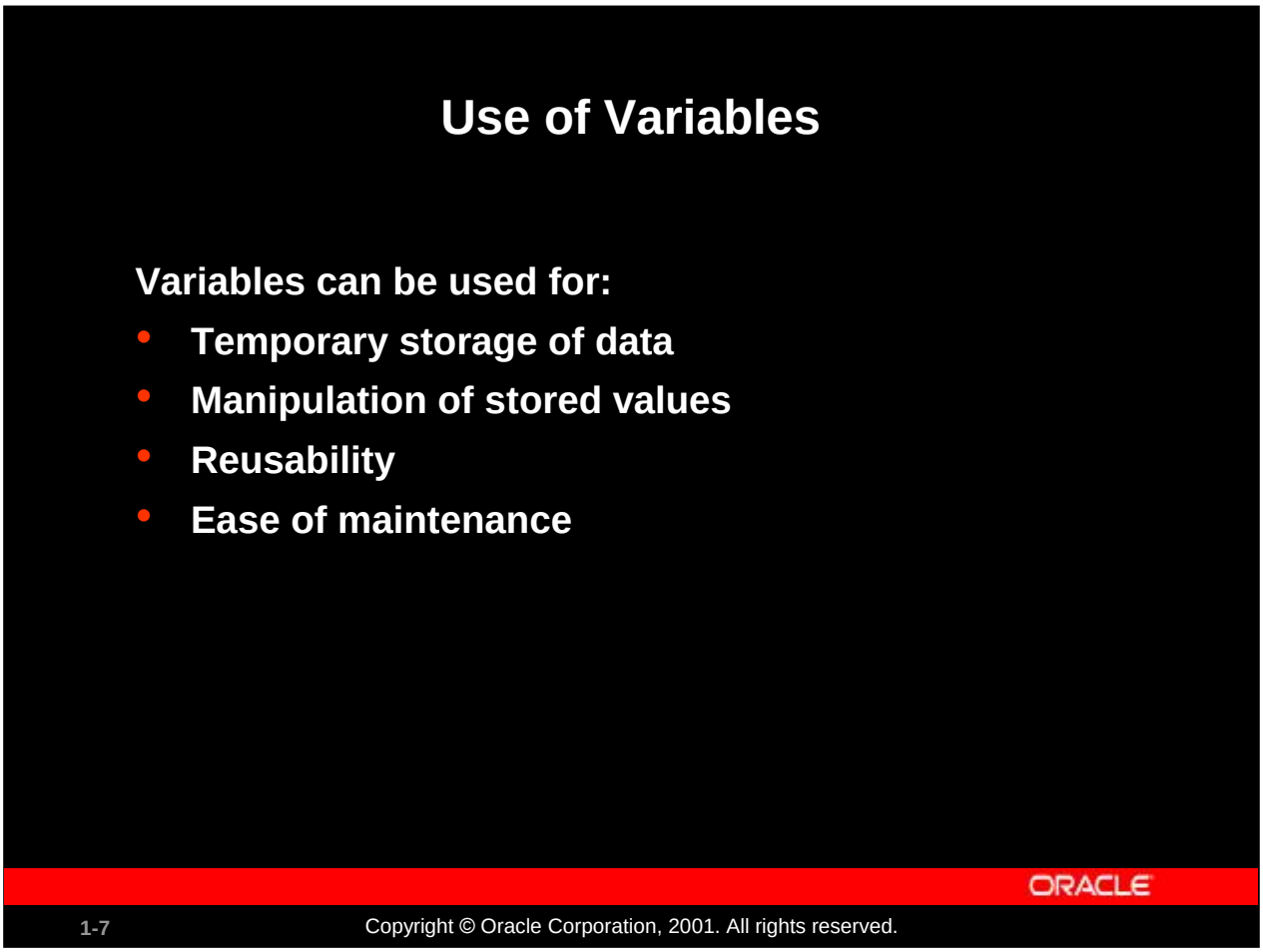
Note: A function is similar to a procedure, except that a function *must* return a value.



Program Constructs

The following table outlines a variety of different PL/SQL program constructs that use the basic PL/SQL block. The program constructs are available based on the environment in which they are executed.

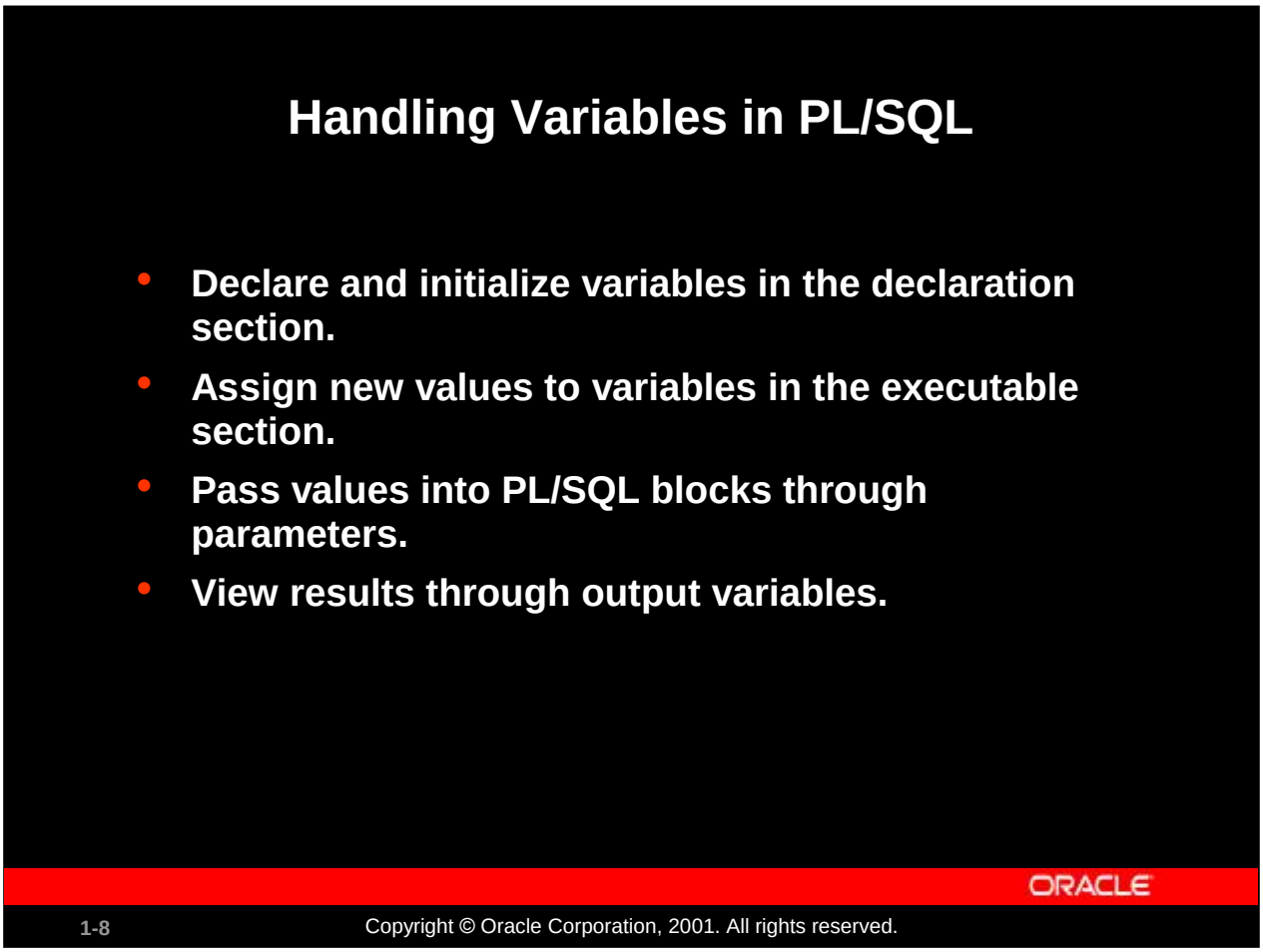
Program Construct	Description	Availability
Anonymous blocks	Unnamed PL/SQL blocks that are embedded within an application or are issued interactively	All PL/SQL environments
Application procedures or functions	Named PL/SQL blocks stored in an Oracle Forms Developer application or shared library; can accept parameters and can be invoked repeatedly by name	Oracle Developer tools components, for example, Oracle Forms Developer, Oracle Reports
Stored procedures or functions	Named PL/SQL blocks stored in the Oracle server; can accept parameters and can be invoked repeatedly by name	Oracle server
Packages (Application or Stored)	Named PL/SQL modules that group related procedures, functions, and identifiers	Oracle server and Oracle Developer tools components, for example, Oracle Forms Developer
Database triggers	PL/SQL blocks that are associated with a database table and fired automatically when triggered by DML statements	Oracle server
Application triggers	PL/SQL blocks that are associated with an application event and fired automatically	Oracle Developer tools components, for example, Oracle Forms Developer
Object types	User-defined composite data types that encapsulate a data structure along with the functions and procedures needed to manipulate the data	Oracle server and Oracle Developer tools



Use of Variables

With PL/SQL you can declare variables and then use them in SQL and procedural statements anywhere that an expression can be used. Variables can be used for the following:

- Temporary storage of data: Data can be temporarily stored in one or more variables for use when validating data input and for processing later in the data flow process.
- Manipulation of stored values: Variables can be used for calculations and other data manipulations without accessing the database.
- Reusability: After they are declared, variables can be used repeatedly in an application simply by referencing them in other statements, including other declarative statements.
- Ease of maintenance: When using %TYPE and %ROWTYPE (more information on %ROWTYPE is covered in a subsequent lesson), you declare variables, basing the declarations on the definitions of database columns. If an underlying definition changes, the variable declaration changes accordingly at run time. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs. More information on %TYPE is covered later in this lesson.



Handling Variables in PL/SQL

Declare and Initialize Variables 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.

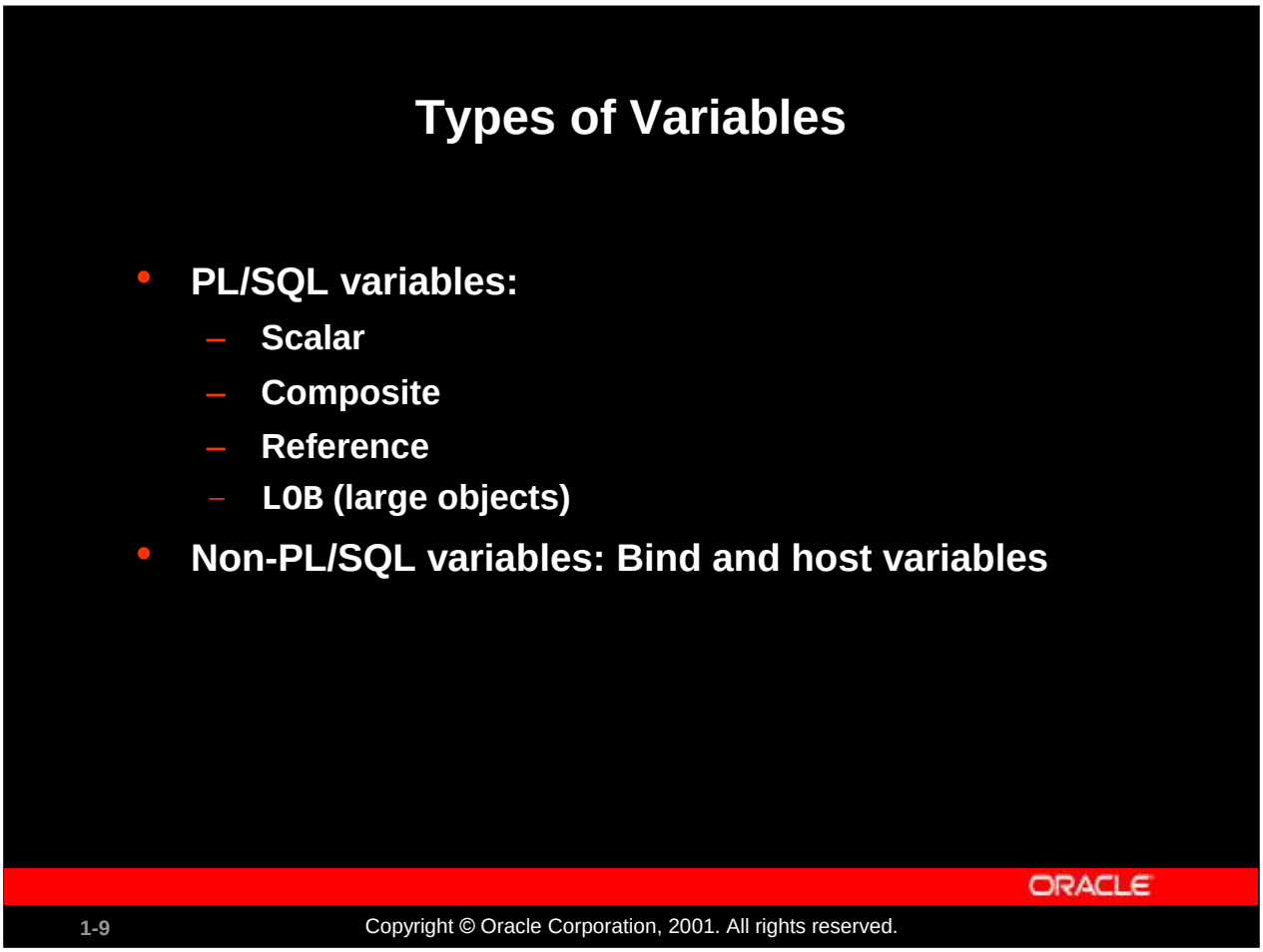
Assign New Values to Variables in the Executable Section

In the executable section, the existing value of the variable is replaced with the new value that is assigned to the variable.

Pass Values Into PL/SQL Subprograms Through Parameters

There are three parameter modes, `IN` (the default), `OUT`, and `IN OUT`. Use the `IN` parameter to pass values to the subprogram being called. Use the `OUT` parameter to return values to the caller of a subprogram. And use the `IN OUT` parameter to pass initial values to the subprogram being called and to return updated values to the caller. We pass values into anonymous block via *i*SQL*PLUS substitution variables.

Note: Viewing the results from a PL/SQL block through output variables is discussed later in the lesson.



Types of Variables

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

- Scalar data types hold a single value. The main data types are those that correspond to column types in Oracle server tables; PL/SQL also supports Boolean variables.
- Composite data types, such as records, allow groups of fields to be defined and manipulated in PL/SQL blocks.
- Reference data types hold values, called pointers, that designate other program items. Reference data types are not covered in this course.
- LOB data types hold values, called locators, that specify the location of large objects (such as graphic images) that are stored out of line. LOB data types are discussed in detail later in this course.

Non-PL/SQL variables include host language variables declared in precompiler programs, screen fields in Forms applications, and *i*SQL*Plus host variables.

For more information on LOBs, see *PL/SQL User's Guide and Reference*, "Fundamentals."

Instructor Note

Appendix D of this course covers REF cursors. You can use it in case of any queries.

Using *i*SQL*Plus Variables Within PL/SQL Blocks

- PL/SQL does not have input or output capability of its own.
- You can reference substitution variables within a PL/SQL block with a preceding ampersand.
- *i*SQL*Plus host (or "bind") variables can be used to pass run time values out of the PL/SQL block back to the *i*SQL*Plus environment.

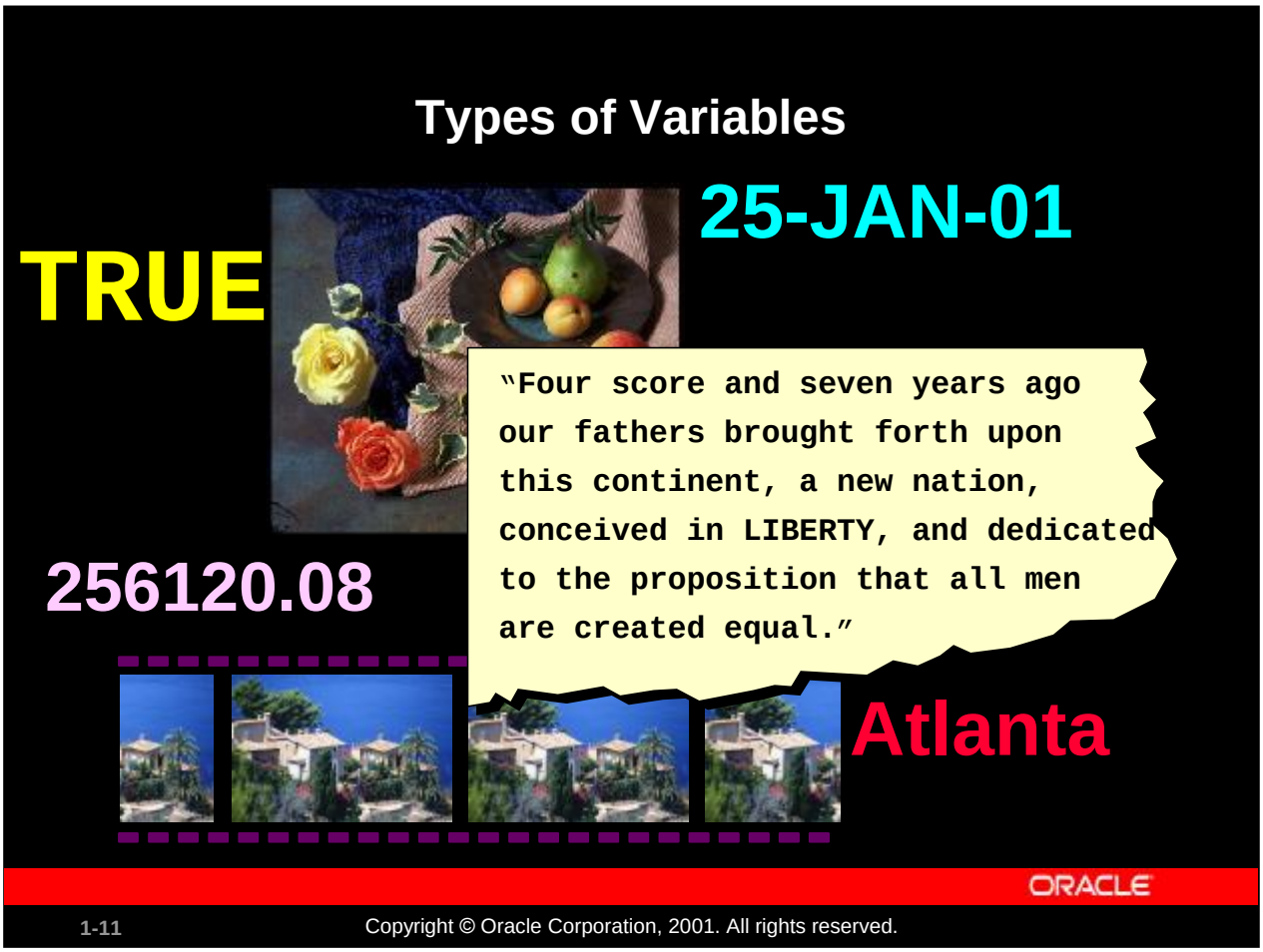
ORACLE

Using *i*SQL*Plus Variables Within PL/SQL Blocks

PL/SQL does not have input or output capability of its own. You must rely on the environment in which PL/SQL is executing to pass values into and out of a PL/SQL block.

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 within a PL/SQL block with a preceding ampersand in the same manner as you reference *i*SQL*Plus substitution variables in a SQL statement. 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. Only one value will replace the substitution variable.

*i*SQL*Plus host variables can be used to pass run-time values out of the PL/SQL block back to the *i*SQL*Plus environment. You can reference host variables in a PL/SQL block with a preceding colon. Bind variables are discussed in further detail later in this lesson.



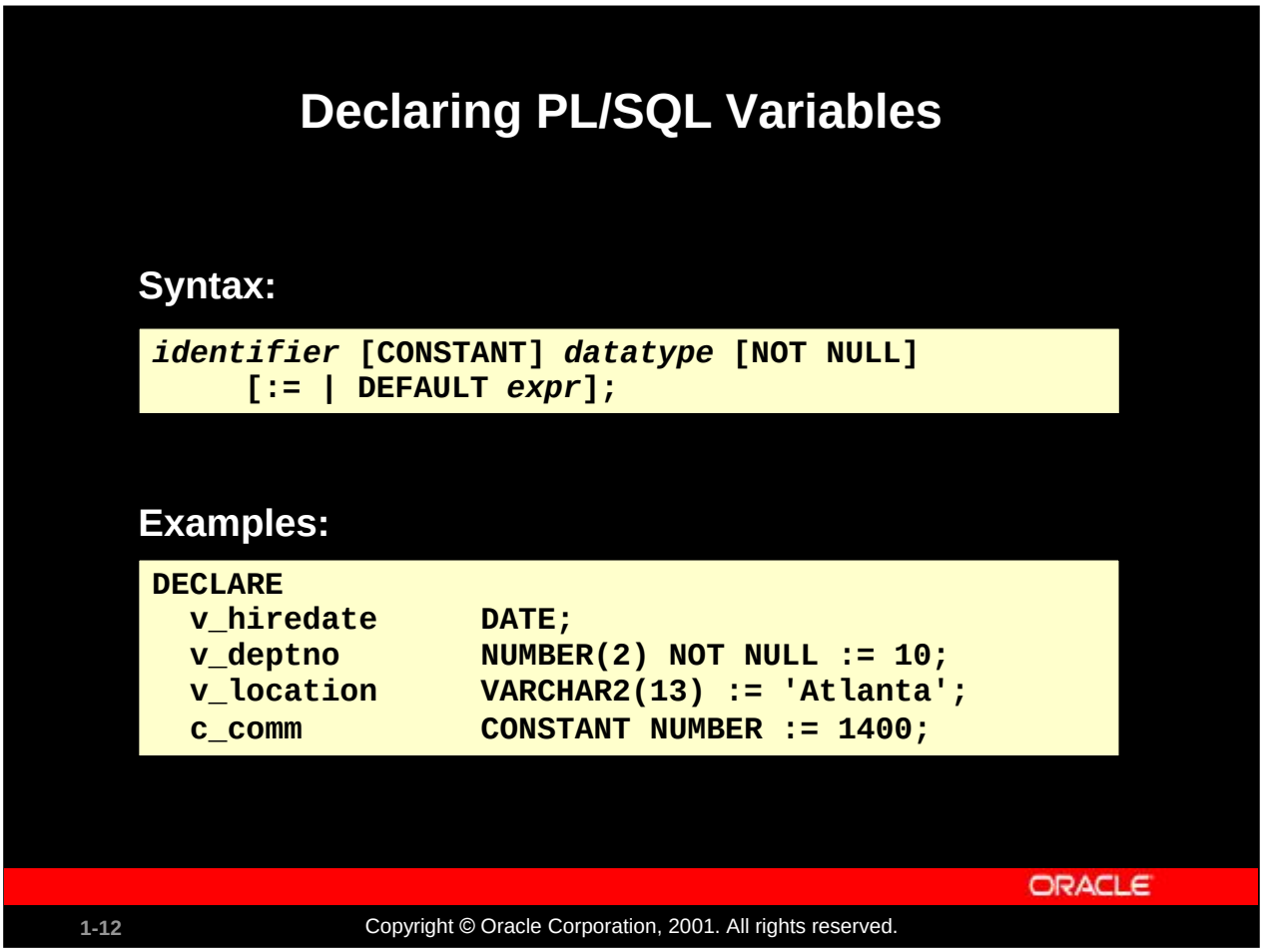
Types of Variables

The slide illustrates the following variable data types:

- TRUE represents a Boolean value.
- 25-JAN-01 represents a DATE.
- The photograph represents a BLOB.
- The text of a speech represents a LONG.
- 256120.08 represents a NUMBER data type with precision and scale.
- The movie represents a BFILE.
- The city name, Atlanta, represents a VARCHAR2.

Instructor Note

Explain that in this lesson and course the terminology identifier and variable will be used interchangeably. Highlight that identifier is the name of the variable.



Declaring PL/SQL Variables

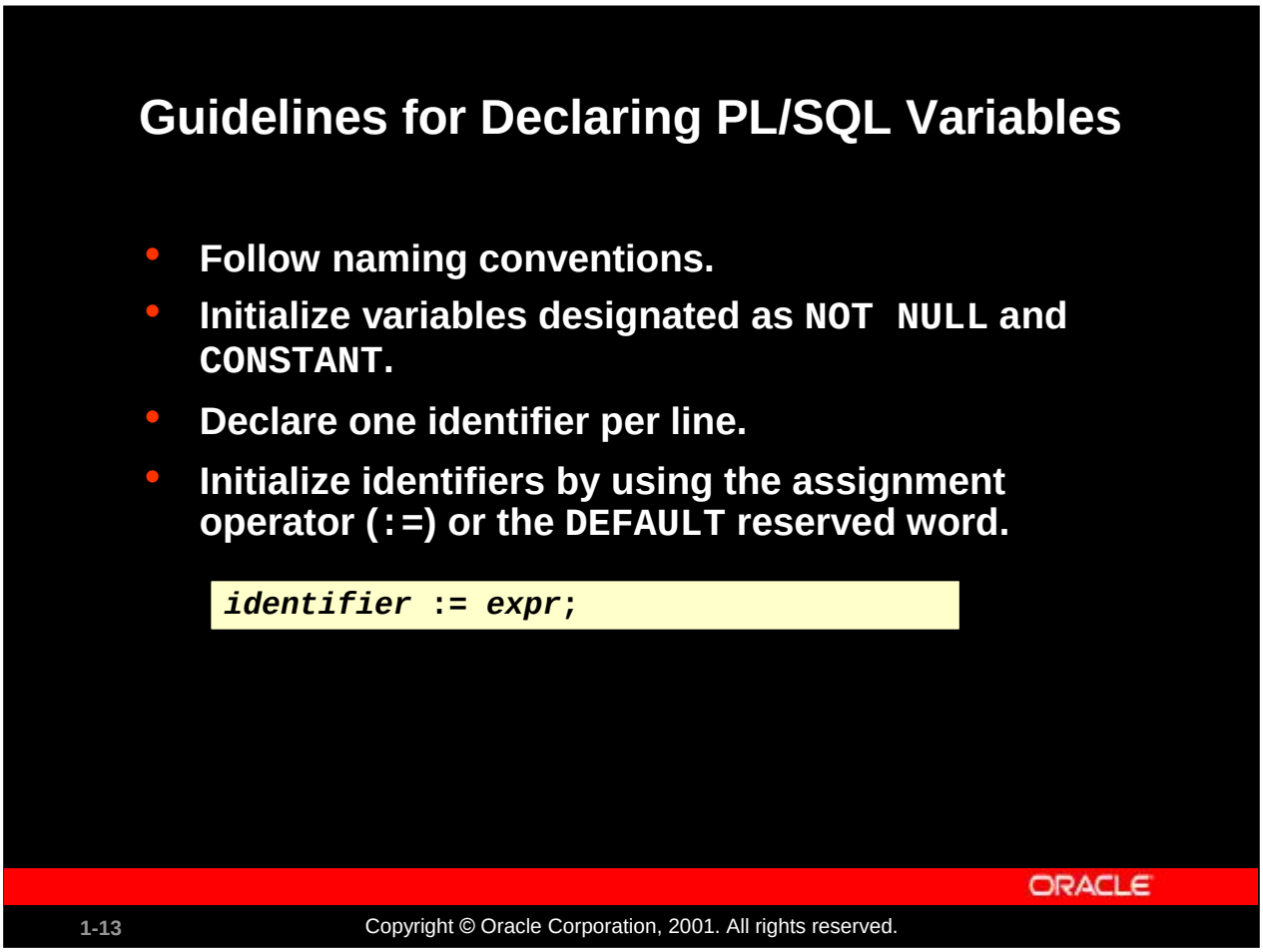
You must declare all PL/SQL identifiers in the declaration section before referencing them in the PL/SQL block. You have the option to assign an initial value to a variable. You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, you must 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.

Instructor Note

Explain that = and := are not the same.



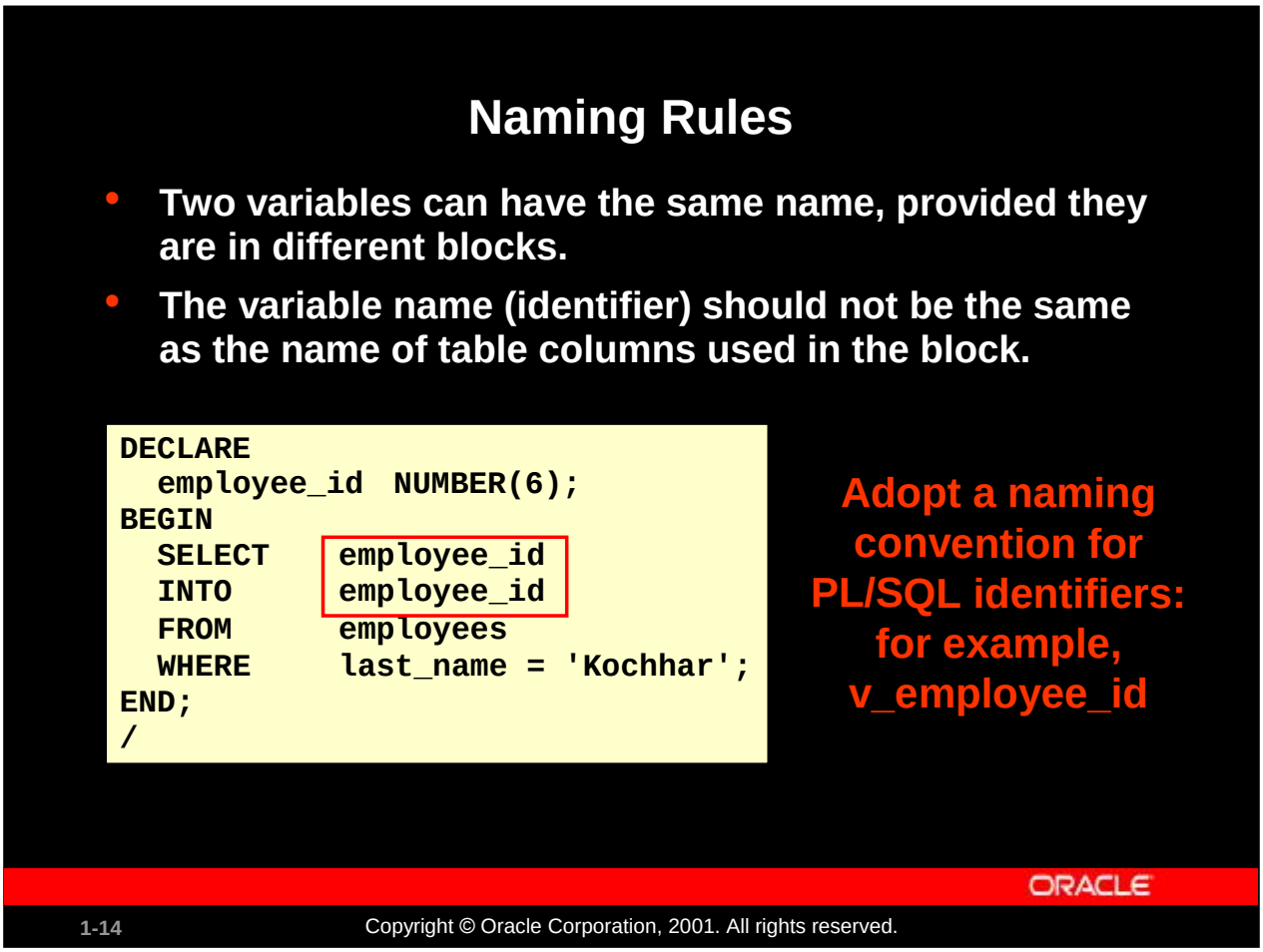
Guidelines for Declaring PL/SQL Variables

Here are some guidelines to follow while declaring PL/SQL variables:

- Name the identifier according to the same rules used for SQL objects.
- You can use naming conventions—for example, *v_name* to represent a variable and *c_name* to represent a constant variable.
- If you use the `NOT NULL` constraint, you must assign a value.
- Declaring only one identifier per line makes code easier to read and maintain.
- 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 50000 to the constant. A constant must be initialized in its declaration; otherwise, you get a compilation error when the declaration is elaborated (compiled).

  ```
  v_sal      CONSTANT REAL := 50000.00;
  ```

- Initialize the variable to an expression with the assignment operator (`:=`) or, equivalently, 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 later. To assign or reassign a value to a variable, you write a PL/SQL assignment statement. You must explicitly name the variable to receive the new value to the left of the assignment operator (`:=`). It is good programming practice to initialize all variables.



Naming Rules

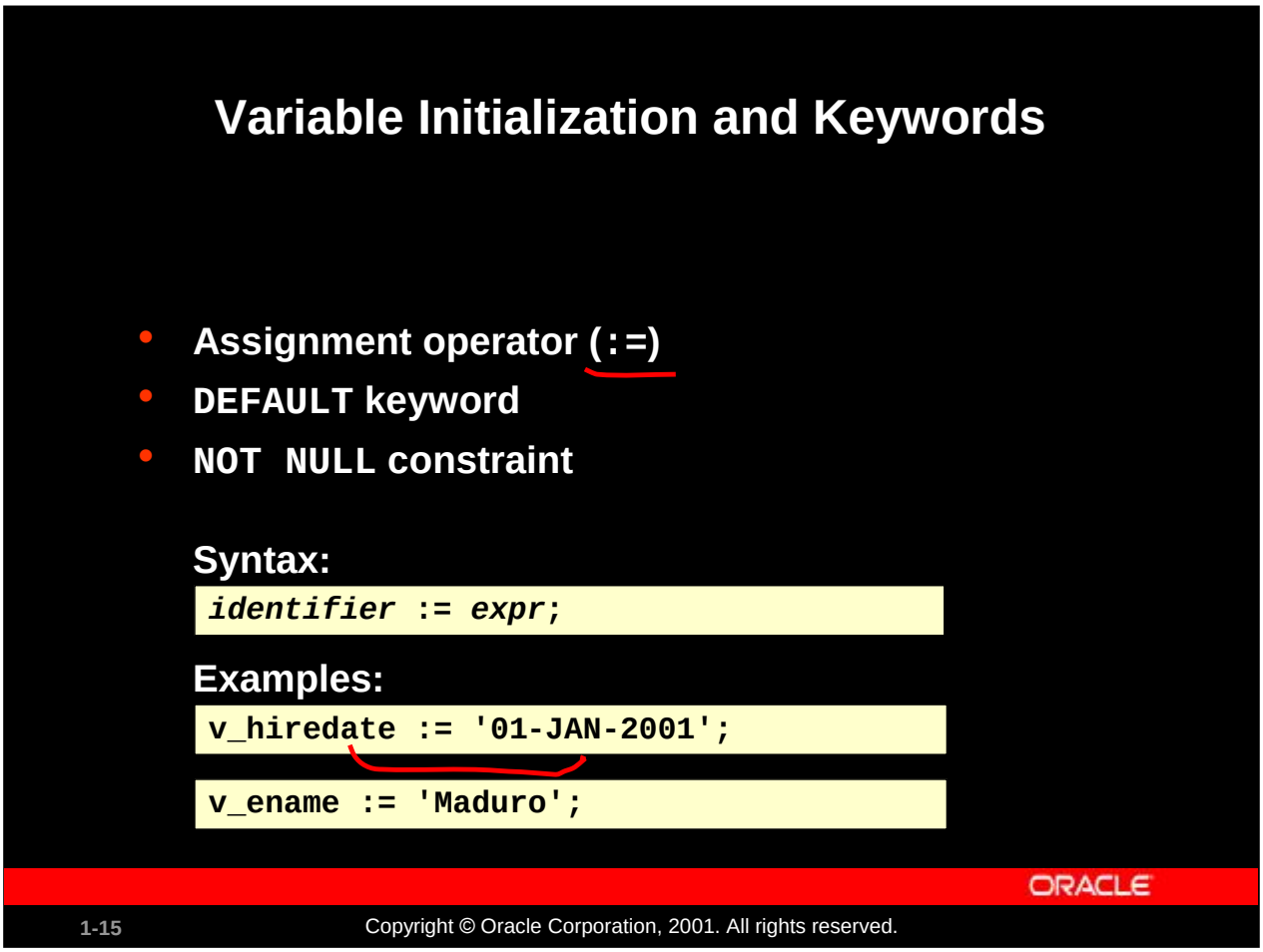
Two objects can have the same name, provided that they are defined in different blocks. Where they coexist, only the object declared in the current block can be used.

You should not choose the same name (identifier) for a variable as the name of table columns used in the block. 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.

Consider adopting a naming convention for various objects that are declared in the DECLARE section of the PL/SQL block. Using v_ as a prefix representing *variable* avoids naming conflicts with database objects.

```
DECLARE
   v_hire_date date;
BEGIN
...
```

Note: The names of the variables must not be longer than 30 characters. The first character must be a letter; the remaining characters can be letters, numbers, or special symbols.



Variable Initialization and Keywords

In the syntax:

identifier	is the name of the scalar variable.
expr	can be a variable, literal, or function call, but *not* a database column.

The variable value assignment examples are defined as follows:

- Set the identifier `V_HIREDATE` to a value of 01-JAN-2001.
- Store the name "Maduro" in the `V_ENAME` identifier.

Variables are initialized every time a block or subprogram is entered. By default, variables are initialized to `NULL`. Unless you explicitly initialize a variable, its value is undefined.

Use the assignment operator (:=) for variables that have no typical value.

```
v_hire_date := '15-SEP-1999'
```

Note: This four-digit value for year, YYYY, assignment is possible only in Oracle8*i* and later. Previous versions may require the use of the `TO_DATE` function.

`DEFAULT`: You can use the `DEFAULT` keyword instead of the assignment operator to initialize variables. Use `DEFAULT` for variables that have a typical value.

```
v_mgr     NUMBER(6) DEFAULT 100;
```

`NOT NULL`: 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.

```
v_city    VARCHAR2(30) NOT NULL := 'Oxford'
```

Variable Initialization and Keywords (continued)

Note: String literals must be enclosed in single quotation marks. For example, `'Hello, world'`. If there is a single quotation mark in the string, use a single quotation mark twice—for example, to insert a value FISHERMAN'S DRIVE, the string would be `'FISHERMAN''S DRIVE'`.

Another way to assign values to variables is to select or fetch database values into it. The following example computes a 10% bonus for the employee with the `EMPLOYEE_ID` 176 and assigns the computed value to the `v_bonus` variable. This is done using the `INTO` clause.

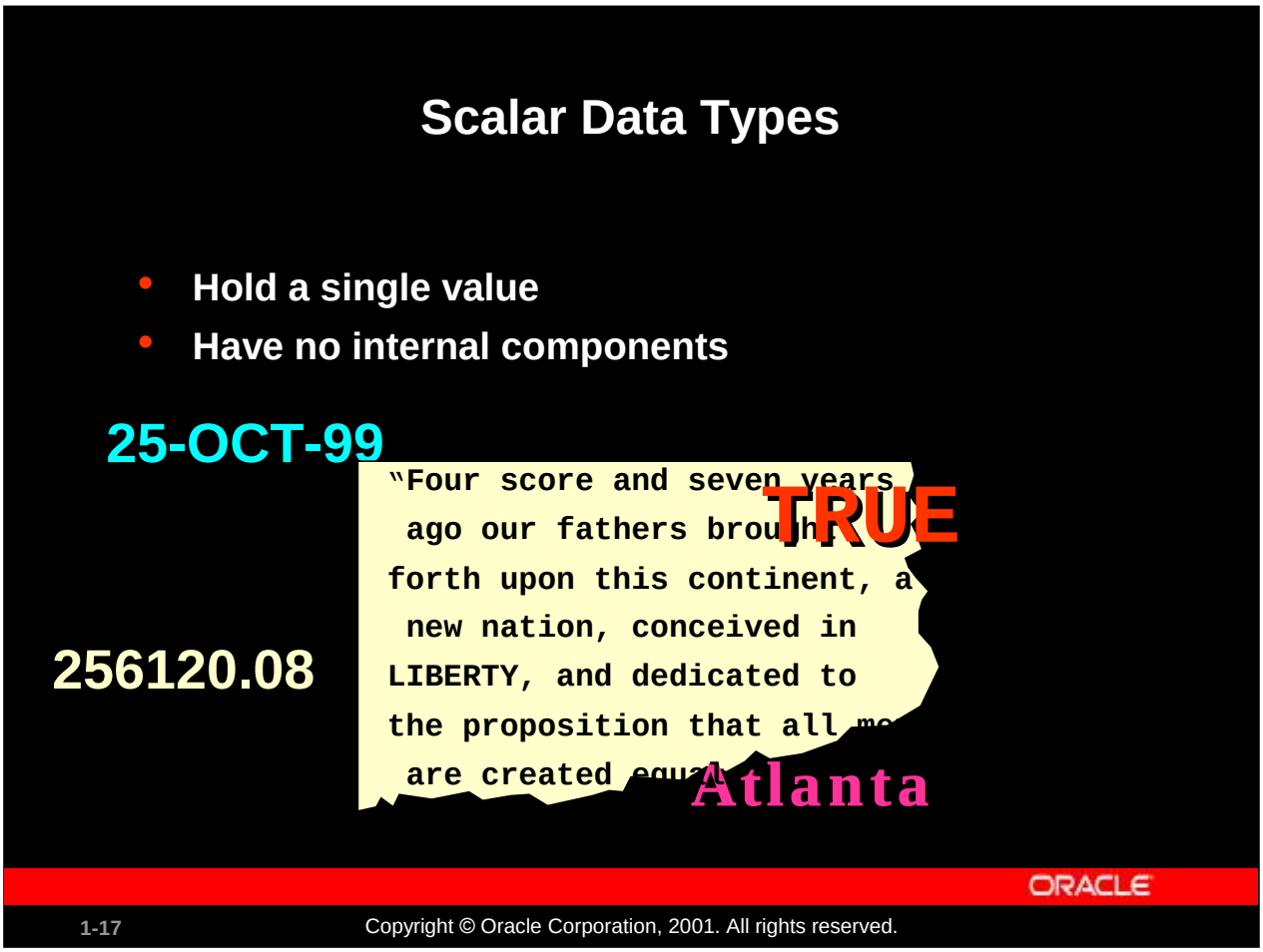
```
DECLARE
    v_bonus NUMBER(8,2);
BEGIN
    SELECT  salary * 0.10
    INTO      v_bonus
    FROM     employees
    WHERE   employee_id = 176;
END;
/
```

Then you can use the variable `v_bonus` in another computation or insert its value into a database table.

Note: To assign a value into a variable from the database, use a `SELECT` or `FETCH` statement. The `FETCH` statement is covered later in this course.

Instructor Note

Explain that you enforce a `NOT NULL` constraint only if it is absolutely necessary. This is because if you enforce a `NOT NULL` constraint, then the Oracle server checks every time for the value to verify whether it is `NULL` or not. This can be a performance overhead.



Scalar Data Types

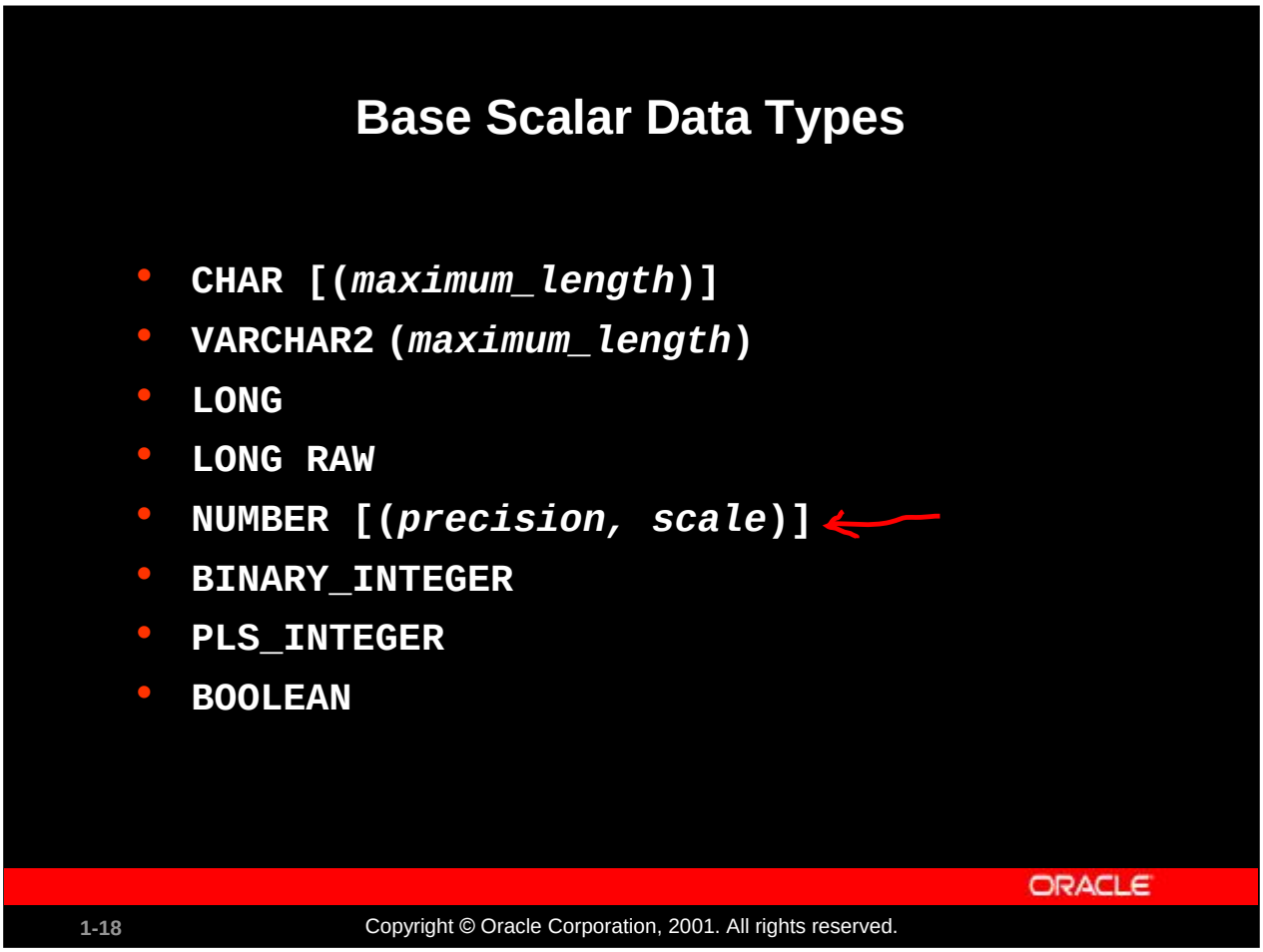
Every constant, variable, and parameter has a data type (or type), which specifies a storage format, constraints, and valid range of values. PL/SQL provides a variety of predefined data types. For instance, you can choose from integer, floating point, character, Boolean, date, collection, reference, and LOB types. In addition, This chapter covers the basic types that are used frequently in PL/SQL programs. Later chapters cover the more specialized types.

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 *PL/SQL User's Guide and Reference*, "Fundamentals."

Instructor Note

Mention that the `NUMBER`, `CHAR`, and `VARCHAR2` data types have subtypes. Additional base types are `RAW` and `ROWID`.



Base Scalar Data Types

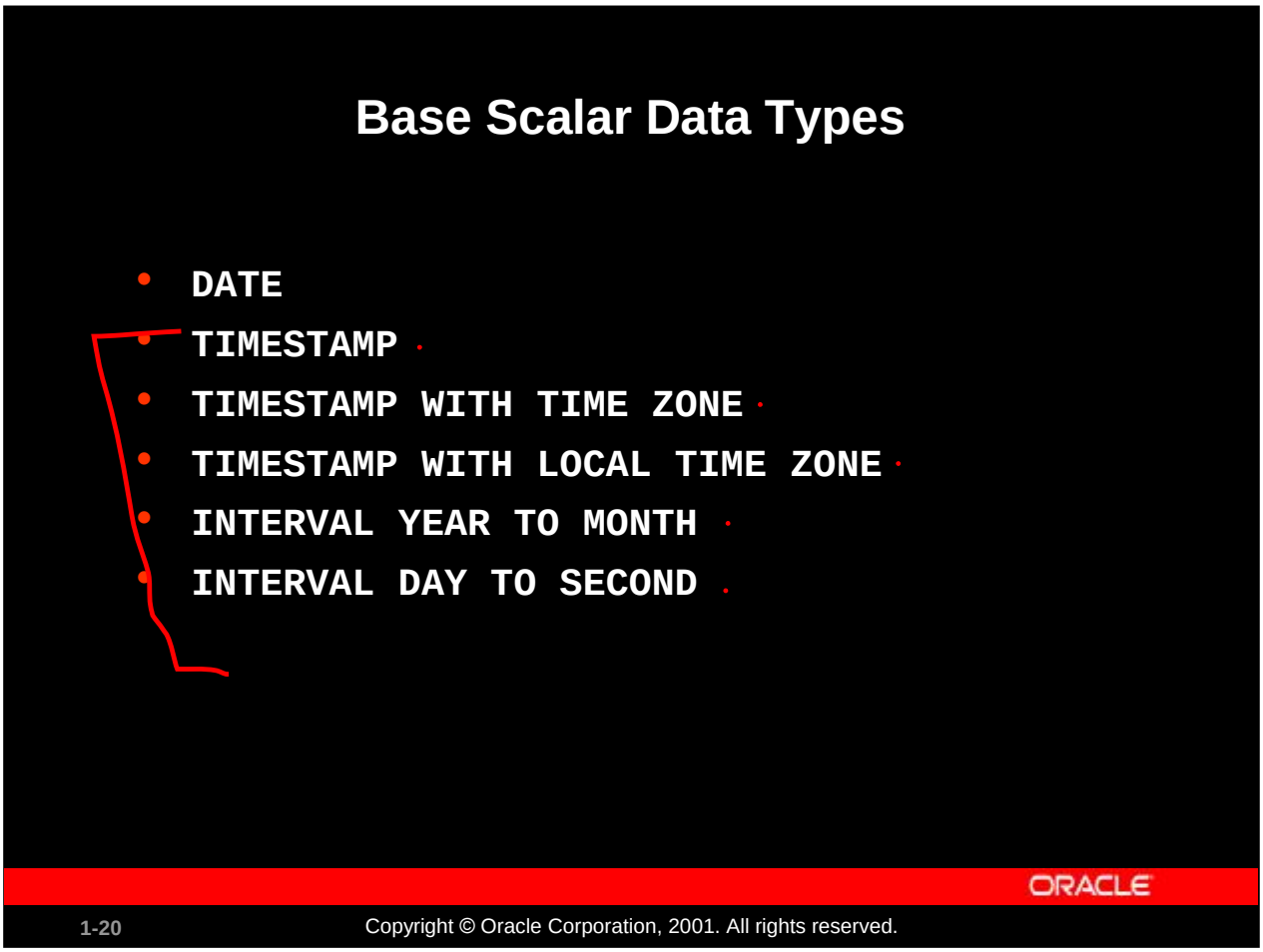
Data Type	Description
CHAR [(*maximum_length*)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a *maximum_length*, the default length is set to 1.
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size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

Base Scalar Data types (continued)

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

Instructor Note

From Oracle9*i*, LOB variables can be used interchangeably with LONG and LONG RAW variables. Oracle recommends migrating any LONG data to the CLOB type, and any LONG RAW data to the BLOB type.



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, and 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 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, Oracle 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 years_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.

Scalar Variable Declarations

Examples:

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

ORACLE

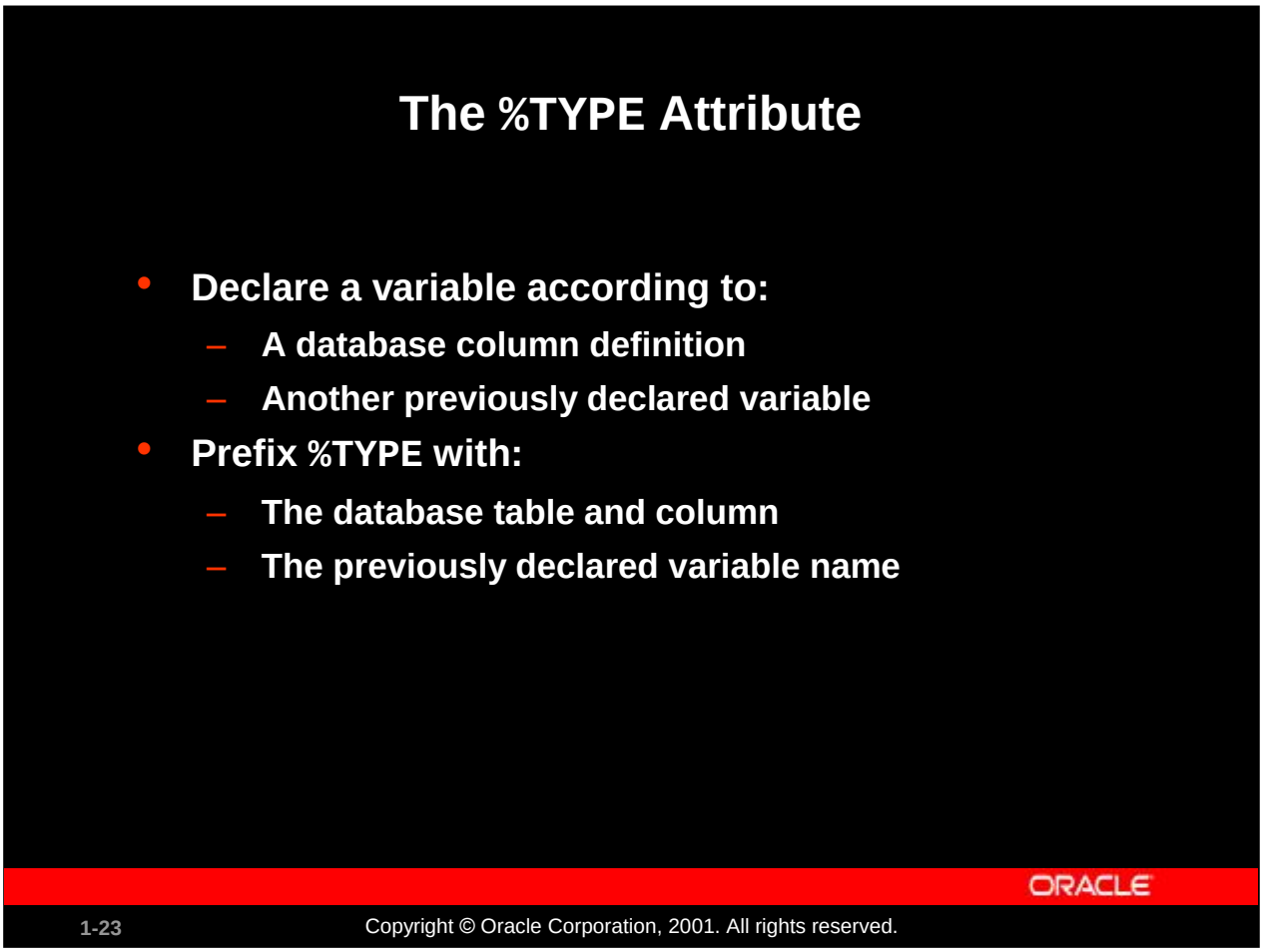
Declaring Scalar Variables

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

- `v_job`: variable to store an employee job title
- `v_count`: variable to count the iterations of a loop and initialized to 0
- `v_total_sal`: variable to accumulate the total salary for a department and initialized to 0
- `v_orderdate`: variable to store the ship date of an order and initialize to one week from today
- `c_tax_rate`: a constant variable for the tax rate, which never changes throughout the PL/SQL block
- `v_valid`: flag to indicate whether a piece of data is valid or invalid and initialized to `TRUE`

Instructor Note

Have a different student explain each declaration in the example on the slide. Students will understand the concept better and it will also break the monotony.



The %TYPE Attribute

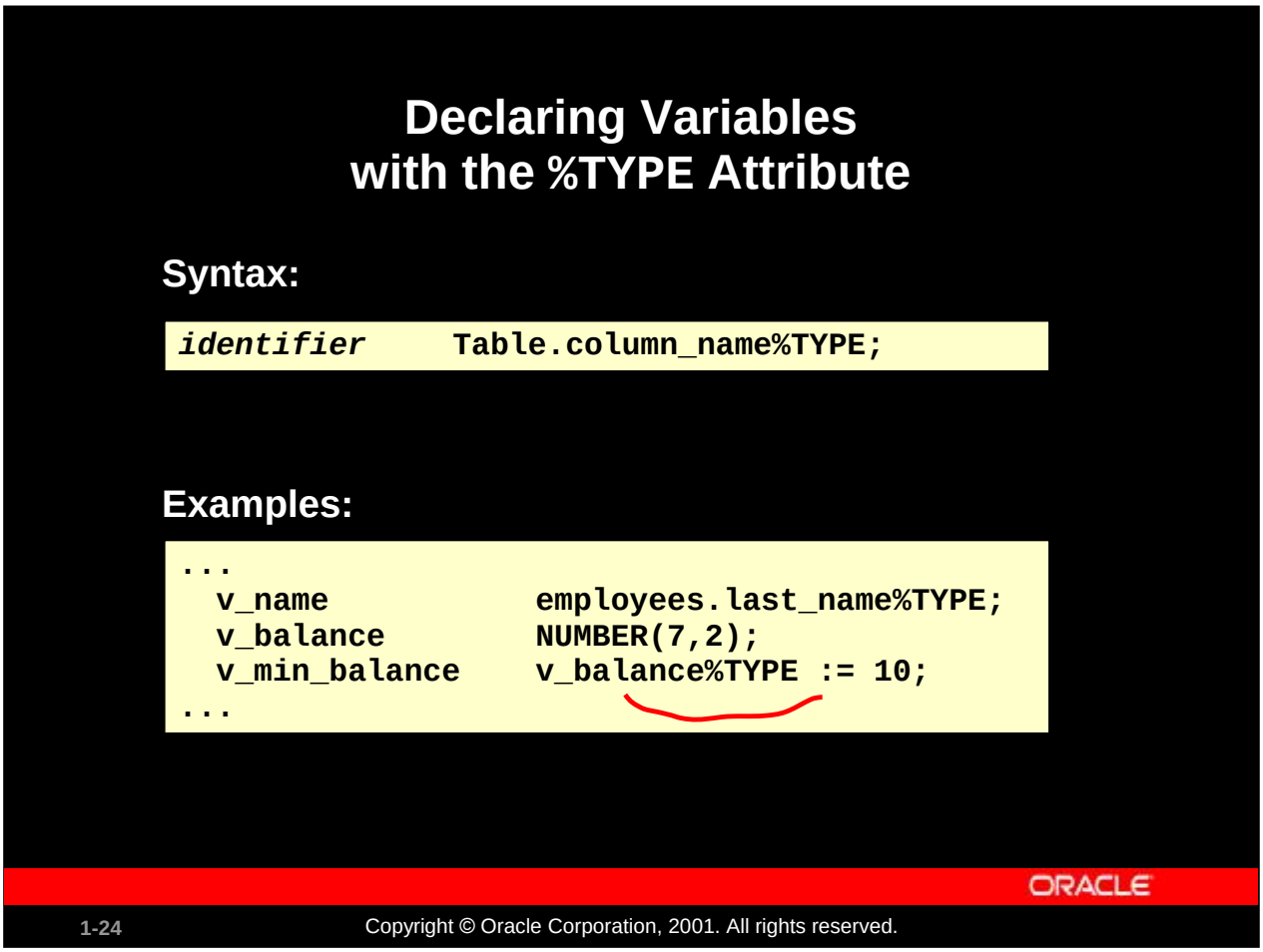
When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct data type and precision. If it is not, a PL/SQL error will occur during execution.

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. To use the attribute in place of the data type that is required in the variable declaration, prefix it with the database table and column name. If referring to a previously declared variable, prefix the variable name to the attribute.

PL/SQL determines the data type and size of the variable when the block is compiled so that such variables are always compatible with the column that is used to populate it. This is a definite advantage for writing and maintaining code, because there is no need to be concerned with column data type changes made at the database level. You can also declare a variable according to another previously declared variable by prefixing the variable name to the attribute.

Instructor Note

The %TYPE attribute has some overhead, in that a SELECT statement is issued against the database to obtain the data type. If the PL/SQL code is in a client tool, the SELECT must be executed each time the block is executed.



Declaring Variables with the %TYPE Attribute

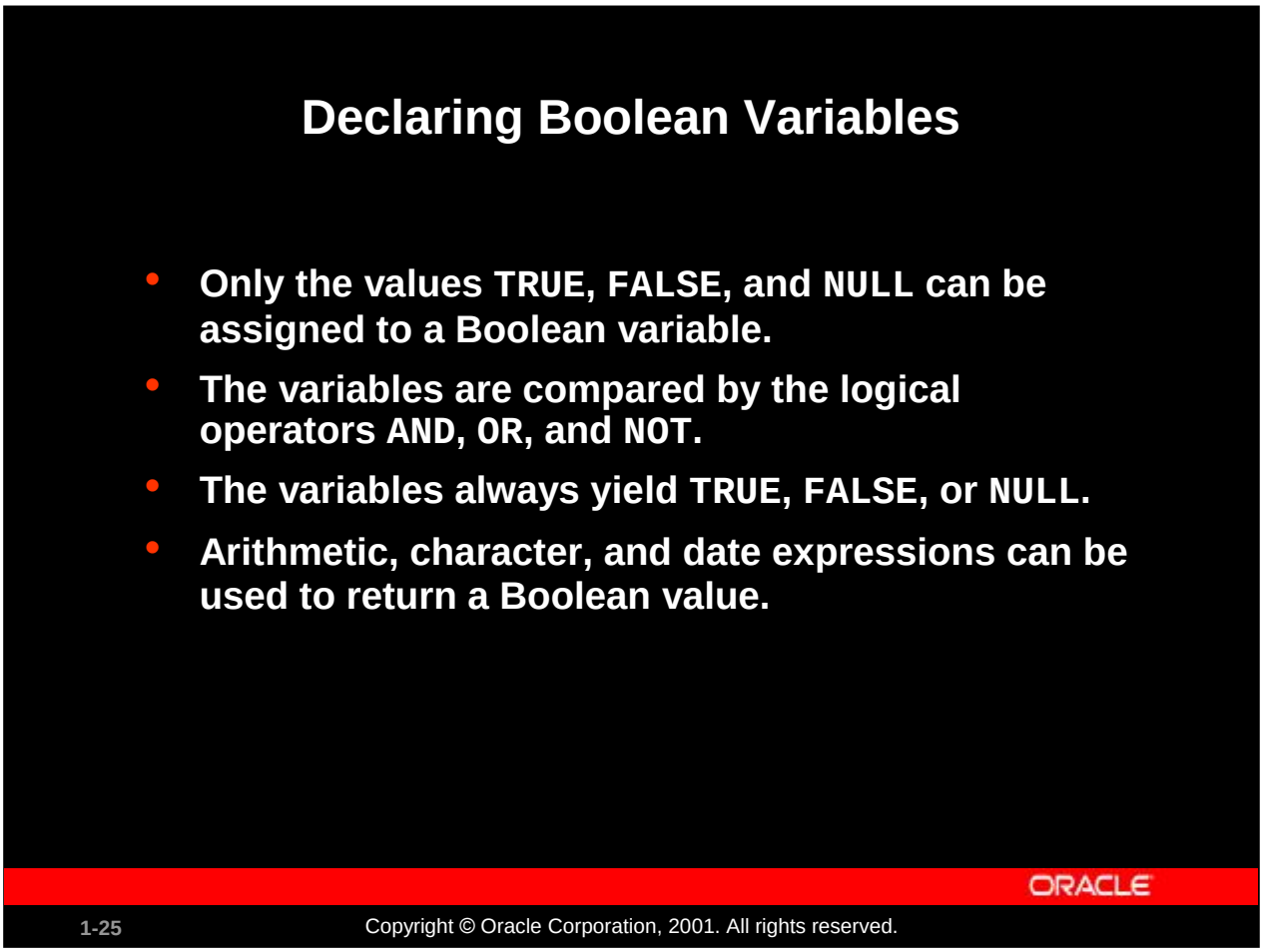
Declare variables to store the last name of an employee. The variable `v_name` is defined to be of the same data type as the `LAST_NAME` column in the `EMPLOYEES` table. `%TYPE` provides the data type of a database column:

```
...
v_name      employees.last_name%TYPE;
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10. The variable `v_min_balance` is defined to be of the same data type as the variable `v_balance`. `%TYPE` provides the data type of a variable:

```
...
v_balance           NUMBER(7,2);
v_min_balance       v_balance%TYPE := 10;
...
```

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

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

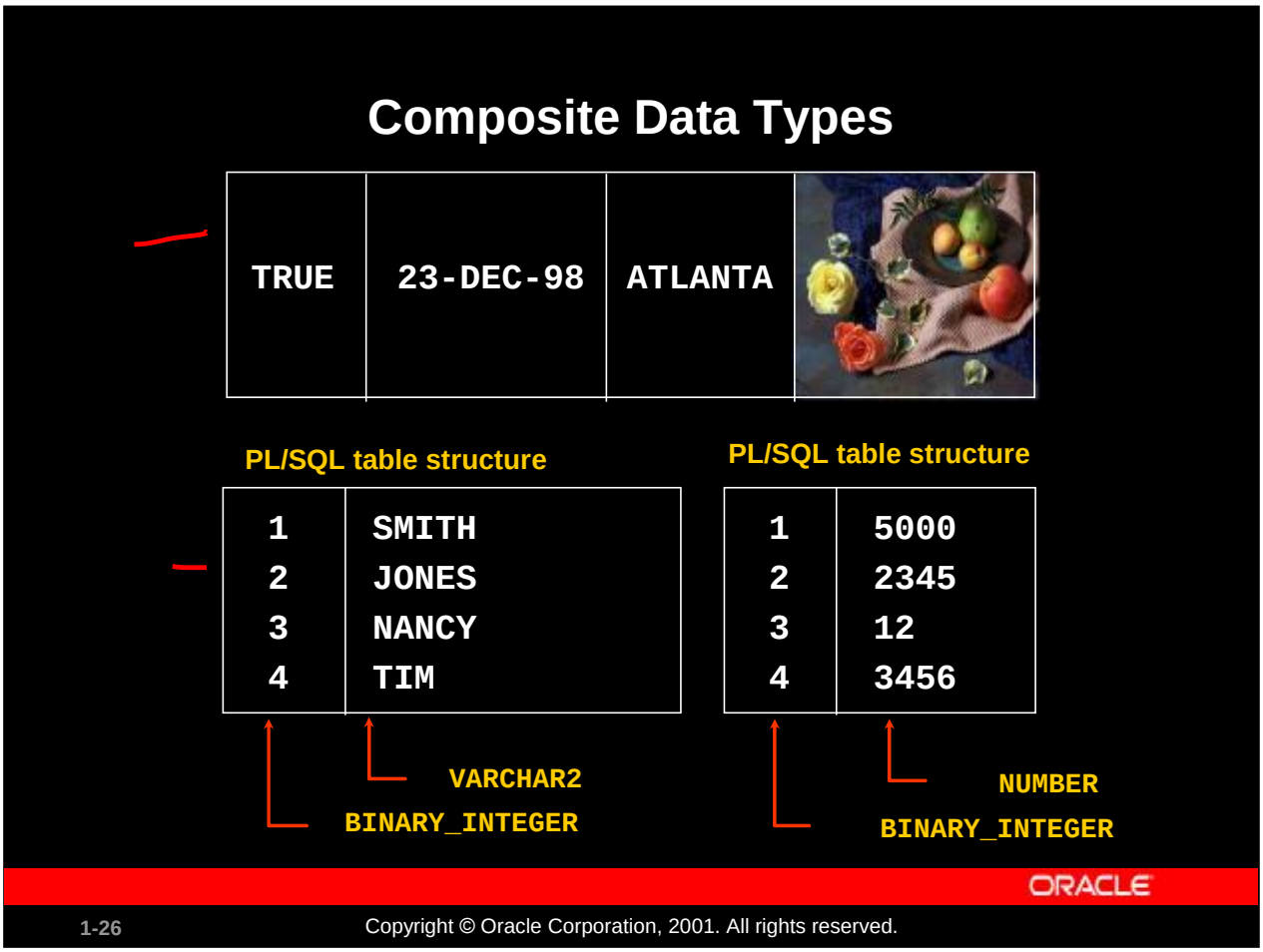
```
v_sal1 := 50000;
v_sal2 := 60000;
```

The following expression yields `TRUE`:

```
v_sal1 < v_sal2
```

Declare and initialize a Boolean variable:

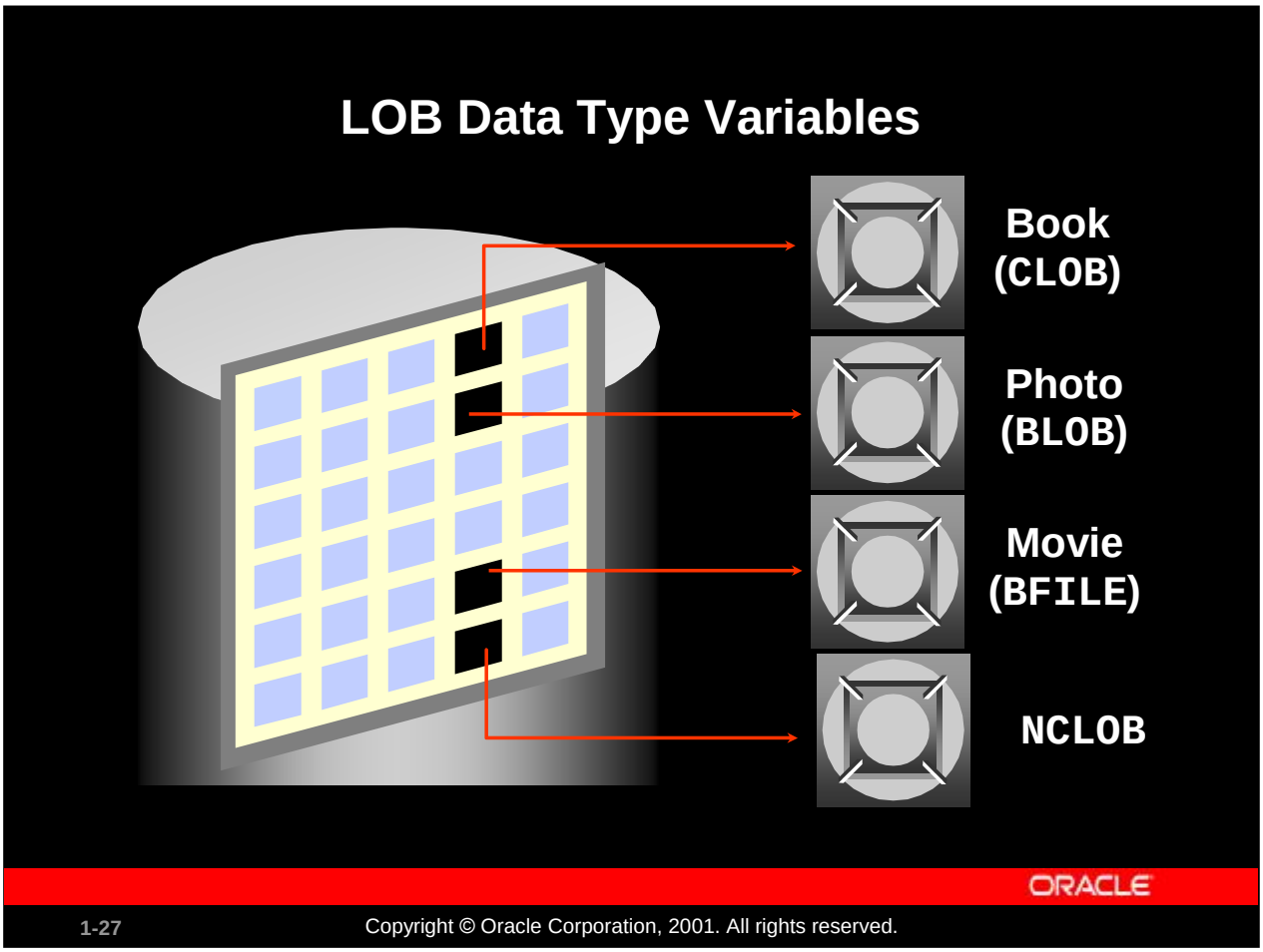
```
DECLARE
    v_flag BOOLEAN := FALSE;
BEGIN
    v_flag := TRUE;
END;
```



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 RECORD data type to treat related but dissimilar data as a logical unit. Use the TABLE data type to reference and manipulate collections of data as a whole object. Both RECORD and TABLE data types are covered in detail in a subsequent lesson. NESTED TABLE and VARRAY data types are covered in the *Advanced PL/SQL* course.

For more information, see *PL/SQL User's Guide and Reference,* "Collections and Records."



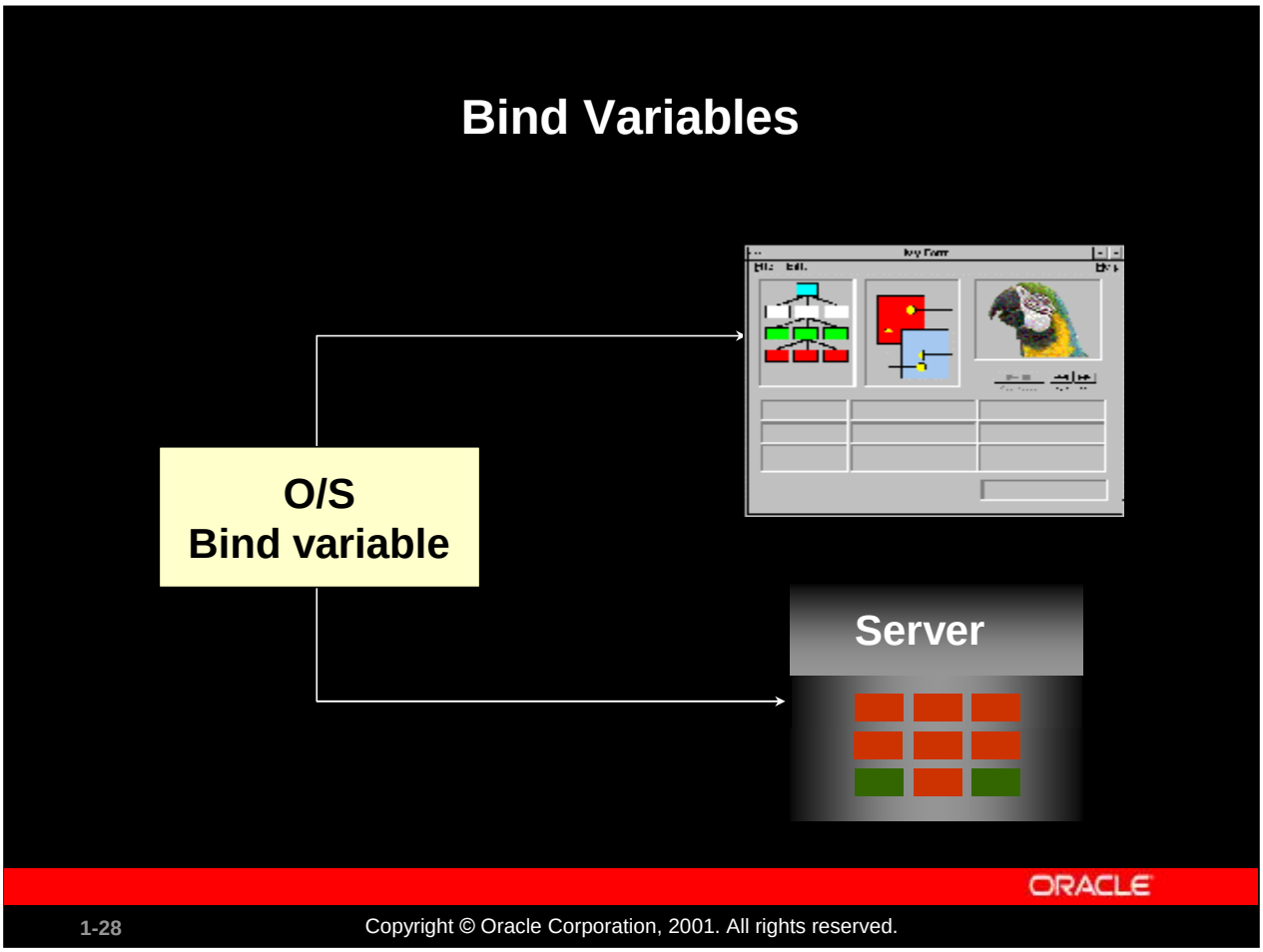
LOB Data Type Variables

With the LOB (large object) data types you can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) up to 4 gigabytes in size. LOB data types allow efficient, random, piecewise access to the data and can be attributes of an object type. LOBs also support random access to data.

- The CLOB (character large object) data type is used to store large blocks of single-byte character data in the database in line (inside the row) or out of line (outside the row).
- The BLOB (binary large object) data type is used to store large binary objects in the database in line (inside the row) or out of line (outside the row).
- The BFILE (binary file) data type is used to store large binary objects in operating system files outside the database.
- The NCLOB (national language character large object) data type is used to store large blocks of single-byte or fixed-width multibyte NCHAR unicode data in the database, in line or out of line.

Instructor Note

The NCLOB data type stores multibyte national character set character (NCHAR) data. Both fixed-width and variable-width character sets are supported. NCLOBs can store up to 4 gigabytes of character text data. NCLOBs have full transactional support. NCLOB value manipulations can be committed and rolled back.



Bind Variables

A bind variable is a variable that you declare in a host environment. Bind variables can be used to pass
run-time values, either number or character, into or out of one or more PL/SQL programs. The PL/SQL programs use bind variables as they would use any other variable. You can reference variables declared in the host or calling environment in PL/SQL statements, unless the statement is in a procedure, function, or package. This includes host language variables declared in precompiler programs, screen fields in Oracle Developer Forms applications, and *i*SQL*Plus bind variables.

Creating Bind Variables

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

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

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

Displaying Bind Variables

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. The following example illustrates a `PRINT` command:

```
VARIABLE g_n NUMBER
...
PRINT g_n
```

You can reference host variables in PL/SQL programs. These variables should be preceded by a colon.

```
VARIABLE RESULT NUMBER
```

An example of using a host variable in a PL/SQL block:

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

Instructor Note

You can find out the result of a PL/SQL program or contents of variables by one of the following methods:

- Store the result in a table and subsequently query the table.
- Use environment variables to store the result.

When a PL/SQL program is written and executed using *i*SQL*Plus, *i*SQL*Plus becomes the host environment for the PL/SQL program. The variables declared in *i*SQL*Plus are called host variables. Then the PL/SQL program is written and executed using, for example, Oracle Forms. Forms becomes a host environment, and variables declared in Oracle Forms are called host variables. Host variables are also called bind variables.

Using Bind Variables

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

Example:

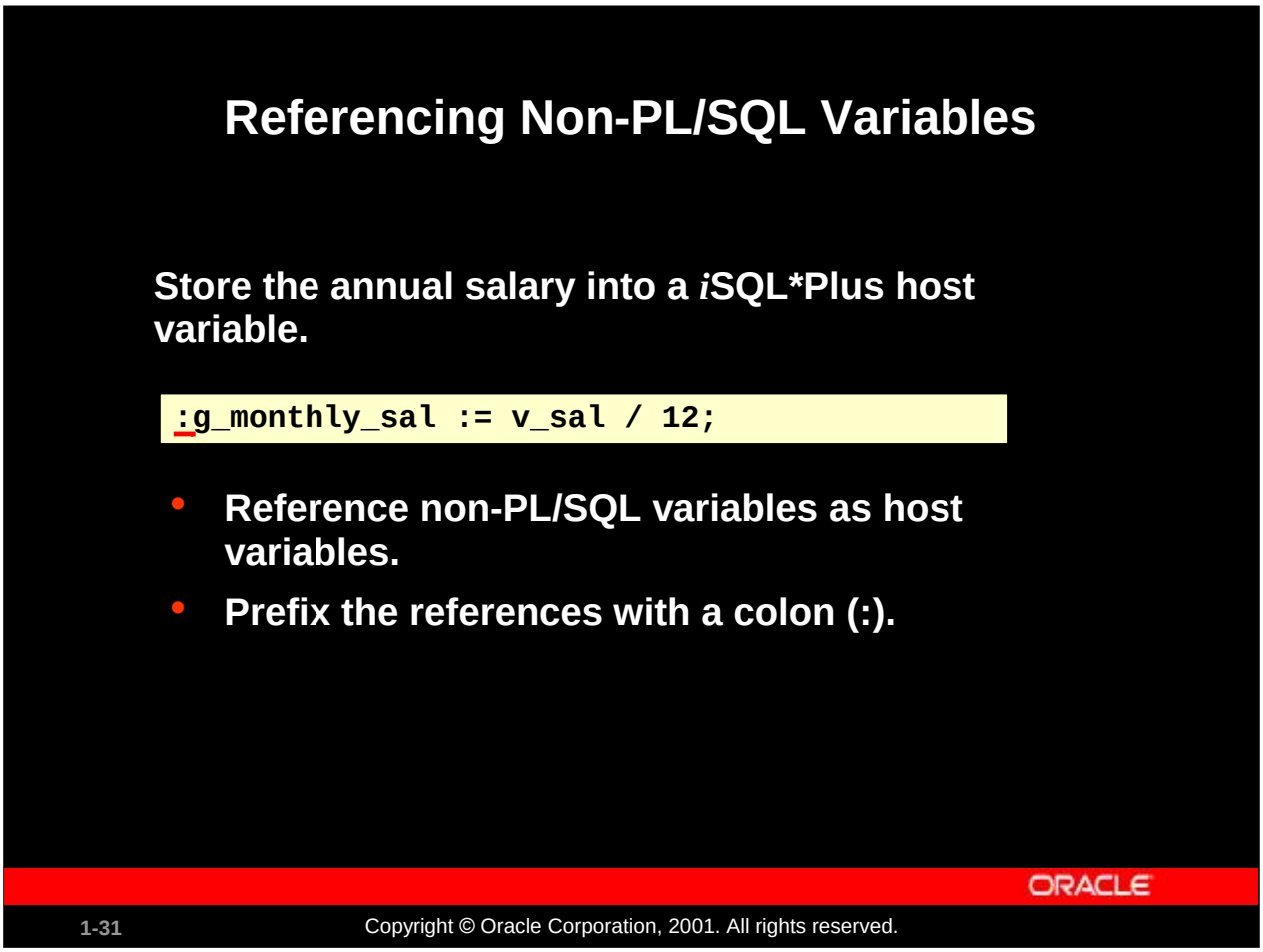
```
VARIABLE      g_salary NUMBER
BEGIN
   SELECT     salary
   INTO       :g_salary
   FROM       employees
   WHERE      employee_id = 178;
END;
/
PRINT g_salary
```

ORACLE

Printing Bind Variables

In *i*SQL*Plus you can display the value of the bind variable using the PRINT command.

G_SALARY
7000



Referencing Non-PL/SQL Variables

To reference host variables, you must prefix the references with a colon (:) to distinguish them from declared PL/SQL variables.

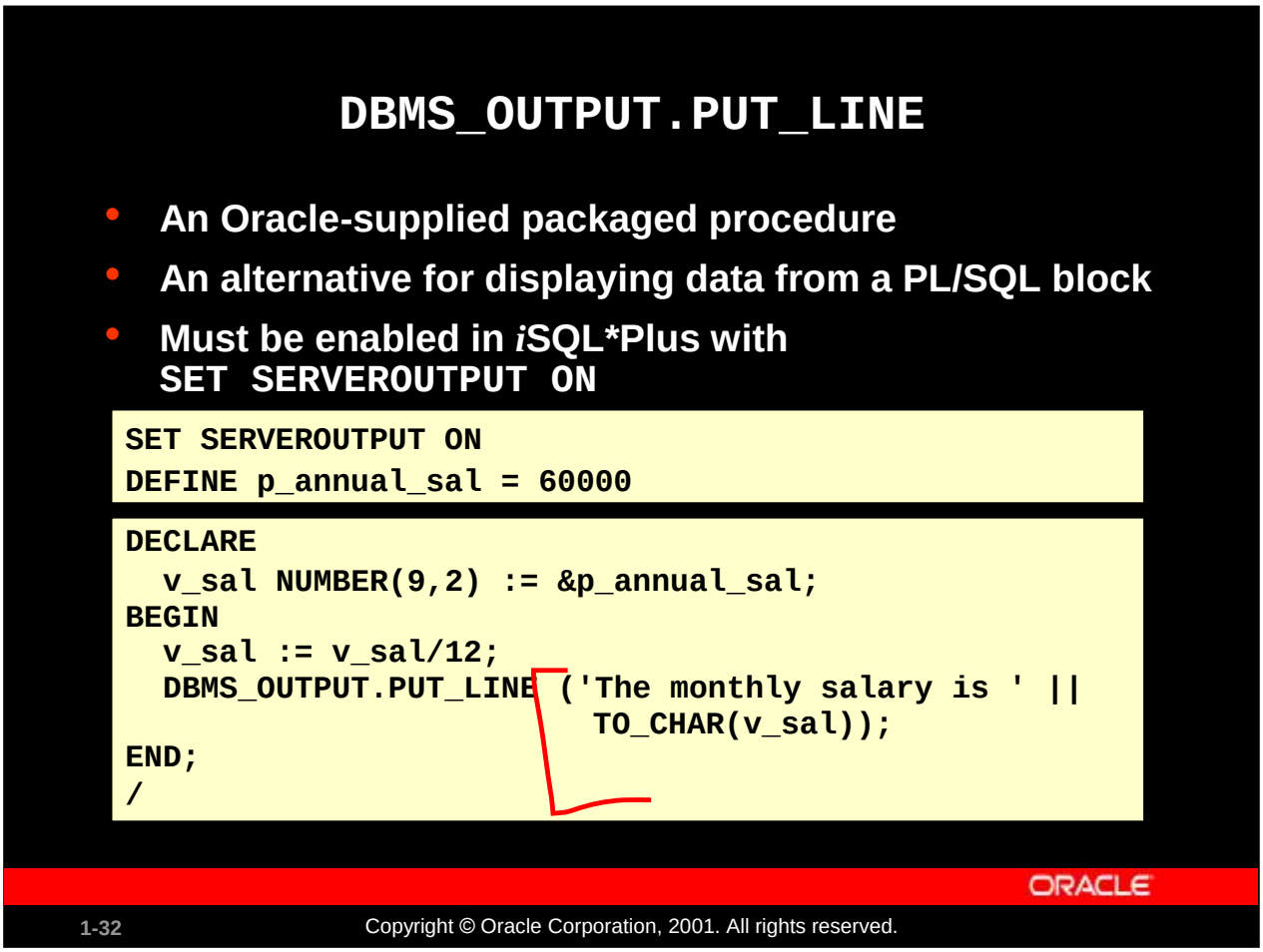
Example

This example computes the monthly salary, based upon the annual salary supplied by the user. This script contains both *i*SQL*Plus commands as well as a complete PL/SQL block.

```
VARIABLE  g_monthly_sal  NUMBER
DEFINE    p_annual_sal = 50000

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

The `DEFINE` command specifies a user variable and assigns it a `CHAR` value. Even though you enter the number 50000, *i*SQL*Plus assigns a `CHAR` value to `p_annual_sal` consisting of the characters, 5,0,0,0 and 0.



DBMS_OUTPUT.PUT_LINE

You have seen that you can declare a host variable, reference it in a PL/SQL block, and then display its contents in *i*SQL*Plus using the `PRINT` command. Another option for displaying information from a PL/SQL block is `DBMS_OUTPUT.PUT_LINE`. `DBMS_OUTPUT` is an Oracle-supplied package, and `PUT_LINE` is a procedure within that package.

Within a PL/SQL block, reference `DBMS_OUTPUT.PUT_LINE` and, in parentheses, specify the string that you want to print to the screen. The package must first be enabled in your *i*SQL*Plus session. To do this, execute the *i*SQL*Plus `SET SERVEROUTPUT ON` command.

The example on the slide computes the monthly salary and prints it to the screen, using `DBMS_OUTPUT.PUT_LINE`. The output is shown below:

```
The monthly salary is 5000
PL/SQL procedure successfully completed.
```

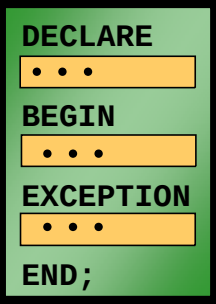
Instructor Note

Mention that the `TO_CHAR` function in the `DBMS_OUTPUT.PUT_LINE` is optional. It is included only for explicit conversion for better performance benefits.

Summary

In this lesson you should have learned that:

- PL/SQL blocks are composed of the following sections:
  - Declarative (optional)
  - Executable (required)
  - Exception handling (optional)
- A PL/SQL block can be an anonymous block, procedure, or function.



ORACLE

Summary

A PL/SQL block is a basic, unnamed unit of a PL/SQL program. It consists of a set of SQL or PL/SQL statements and it performs a single 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. The executable part is the mandatory part of a PL/SQL block, and contains SQL and PL/SQL statements for querying and manipulating data. The exception-handling part is embedded inside the executable part of a block and is placed at the end of the executable part.

An anonymous PL/SQL block is the basic, unnamed unit of a PL/SQL program. Procedures and functions can be compiled separately and stored permanently in an Oracle database, ready to be executed.

Summary

In this lesson you should have learned that:

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

ORACLE

Summary (continued)

All PL/SQL data types are scalar, composite, reference, or LOB type. Scalar data types do not have any components within them, whereas composite data types have other data types within them. PL/SQL variables are declared and initialized in the declarative section.

When a PL/SQL program is written and executed using *i*SQL*Plus, *i*SQL*Plus becomes the host environment for the PL/SQL program. The variables declared in *i*SQL*Plus are called host variables. Then the PL/SQL program is written and executed using, for example, Oracle Forms. Forms becomes a host environment, and variables declared in Oracle Forms are called host variables. Host variables are also called bind variables.

To display information from a PL/SQL block use `DBMS_OUTPUT.PUT_LINE`. `DBMS_OUTPUT` is an Oracle-supplied package, and `PUT_LINE` is a procedure within that package. Within a PL/SQL block, reference `DBMS_OUTPUT.PUT_LINE` and, in parentheses, specify the string that you want to print to the screen.

Practice 1 Overview

This practice covers the following topics:

- Determining validity of declarations
- Declaring a simple PL/SQL block
- Executing a simple PL/SQL block

ORACLE

Practice 1 Overview

This practice reinforces the basics of PL/SQL covered in this lesson, including data types, definitions of identifiers, and validation of expressions. You put all these elements together to create a simple PL/SQL block.

Paper-Based Questions

Questions 1 and 2 are paper-based questions.

Practice 1

1. Evaluate each of the following declarations. Determine which of them are *not* legal and explain why.

a.
```
DECLARE
  v_id                  NUMBER(4);
```

b.
```
DECLARE
  v_x, v_y, v_z         VARCHAR2(10);
```

c.
```
DECLARE
  v_birthdate           DATE NOT NULL;
```

d.
```
DECLARE
  v_in_stock            BOOLEAN := 1;
```

Practice 1 (continued)

2. In each of the following assignments, indicate whether the statement is valid and what the valid data type of the result will be.

   a. `v_days_to_go := v_due_date - SYSDATE;`

   b. `v_sender := USER || ': ' || TO_CHAR(v_dept_no);`

   c. `v_sum := $100,000 + $250,000;`

   d. `v_flag := TRUE;`

   e. `v_n1 := v_n2 > (2 * v_n3);`

   f. `v_value := NULL;`

3. Create an anonymous block to output the phrase "My PL/SQL Block Works" to the screen.

G_MESSAGE
My PL/SQL Block Works

Practice 1 (continued)

If you have time, complete the following exercise:

4. Create a block that declares two variables. Assign the value of these PL/SQL variables to *i*SQL*Plus host variables and print the results of the PL/SQL variables to the screen. Execute your PL/SQL block. Save your PL/SQL block in a file named `p1q4.sql`, by clicking the `Save Script` button. Remember to save the script with a `.sql` extension.

```
V_CHAR          Character (variable length)
V_NUM           Number
```

Assign values to these variables as follows:

```
Variable Value
-------- -------------------------------------

V_CHAR   The literal '42 is the answer'
V_NUM    The first two characters from V_CHAR
```

G_CHAR
42 is the answer

G_NUM
42