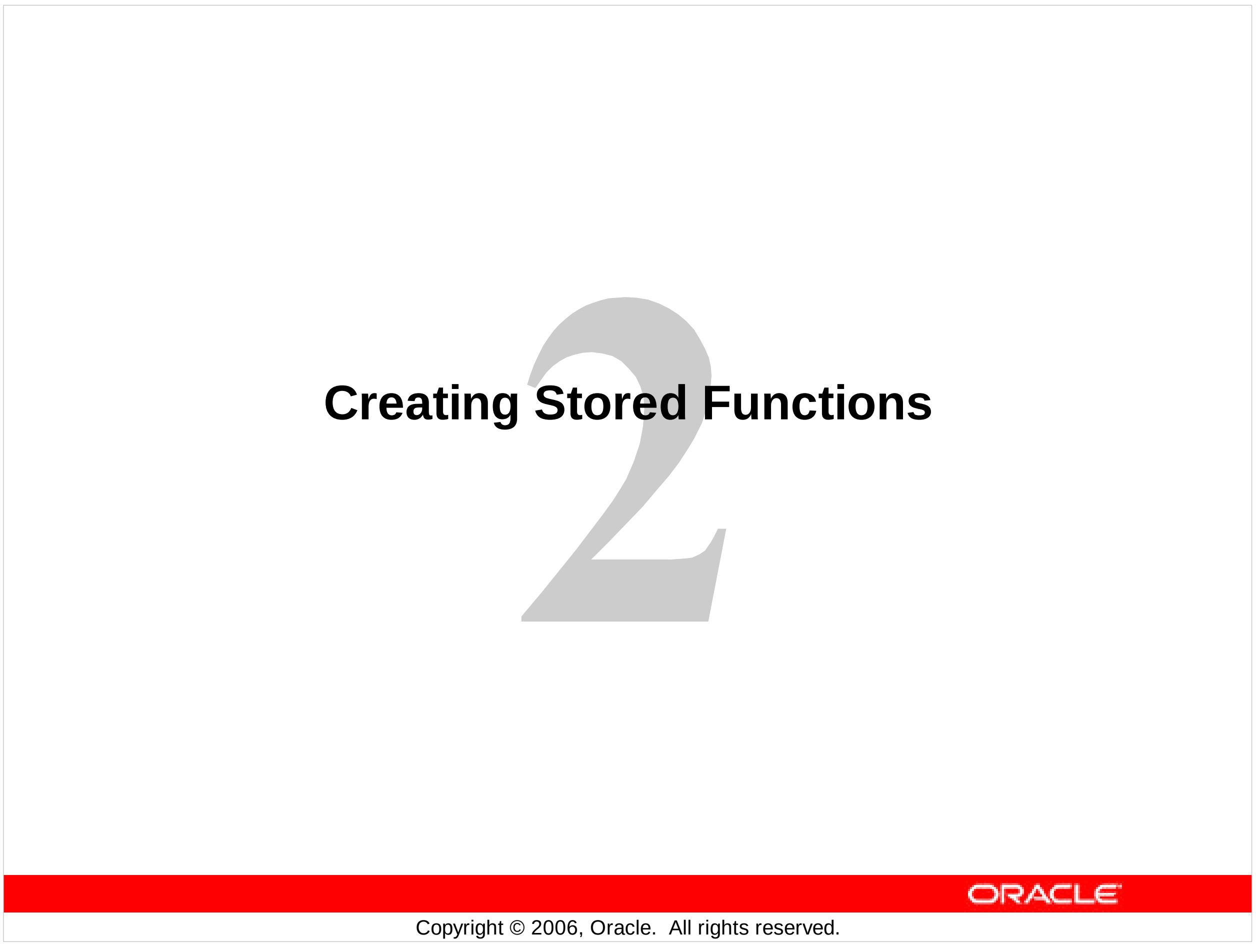


2

Creating Stored Functions

Objectives

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

- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Differentiate between a procedure and a function

Lesson Aim

In this lesson, you learn how to create and invoke functions.

Overview of Stored Functions

A function:

- Is a named PL/SQL block that returns a value
- Can be stored in the database as a schema object for repeated execution
- Is called as part of an expression or is used to provide a parameter value

Overview of Stored Functions

A function is a named PL/SQL block that can accept parameters, be invoked, and return a value. In general, you use a function to compute a value. Functions and procedures are structured alike. A function must return a value to the calling environment, whereas a procedure returns zero or more values to its calling environment. Like a procedure, a function has a header, a declarative section, an executable section, and an optional exception-handling section. A function must have a `RETURN` clause in the header and at least one `RETURN` statement in the executable section.

Functions can be stored in the database as schema objects for repeated execution. A function that is stored in the database is referred to as a stored function. Functions can also be created on client-side applications.

Functions promote reusability and maintainability. When validated, they can be used in any number of applications. If the processing requirements change, only the function needs to be updated.

A function may also be called as part of a SQL expression or as part of a PL/SQL expression. In the context of a SQL expression, a function must obey specific rules to control side effects. In a PL/SQL expression, the function identifier acts like a variable whose value depends on the parameters passed to it.

Syntax for Creating Functions

The PL/SQL block must have at least one `RETURN` statement.

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, ...)]
RETURN datatype IS|AS
 [local_variable_declarations; …]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

PL/SQL Block

2-4

Syntax for Creating Functions

A function is a PL/SQL block that returns a value. A `RETURN` statement must be provided to return a value with a data type that is consistent with the function declaration.

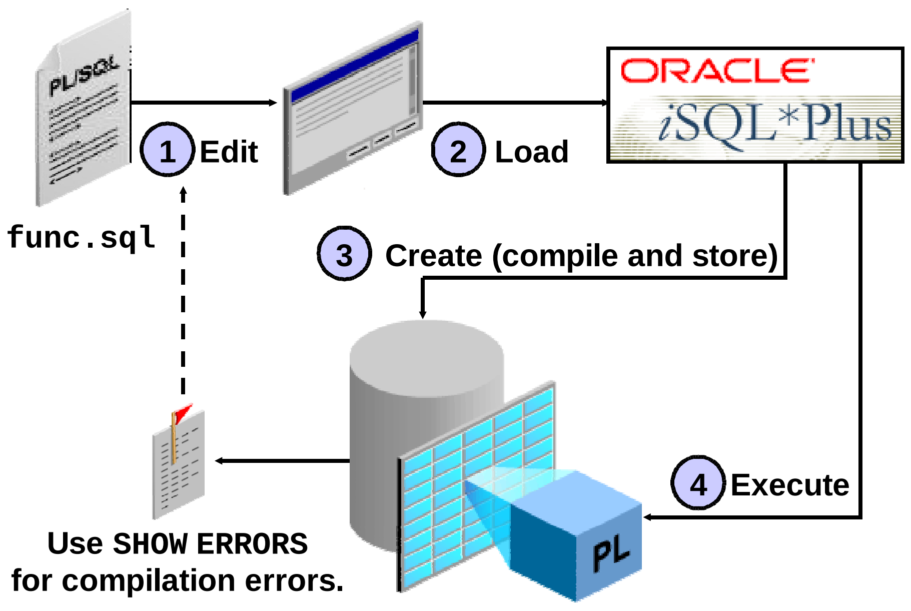
You create new functions with the `CREATE FUNCTION` statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

You should consider the following points about the `CREATE FUNCTION` statement:

- The `REPLACE` option indicates that if the function exists, it is dropped and replaced with the new version that is created by the statement.
- The `RETURN` data type must not include a size specification.
- The PL/SQL block starts with a `BEGIN` after the declaration of any local variables and ends with an `END`, optionally followed by the *`function_name`*.
- There must be at least one `RETURN` *`expression`* statement.
- You cannot reference host or bind variables in the PL/SQL block of a stored function.

Note: Although the `OUT` and `IN OUT` parameter modes can be used with

Developing Functions



How to Develop Stored Functions

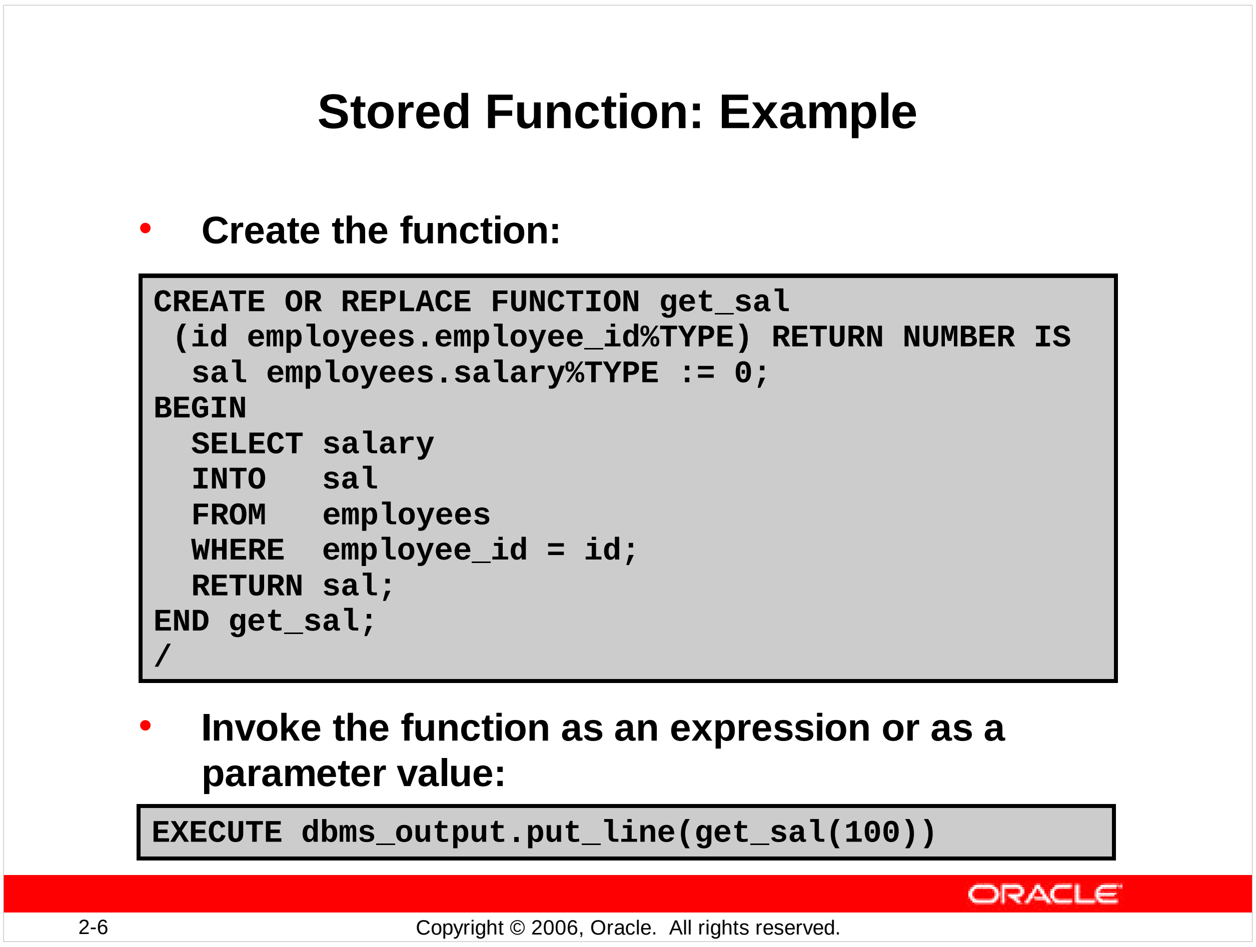
The diagram illustrates the basic steps involved in developing a stored function. To develop a stored function, perform the following steps:

1. Create a file by using your favorite text or code editor to edit the function syntax, and saving the code in a file typically with a `.sql` extension.
2. Load the function code from the file into the buffer by using *i*SQL*Plus as the PL/SQL development environment.
3. Execute the `CREATE FUNCTION` statement to compile and store the function in the database.
4. After successful compilation, invoke the function from a PL/SQL environment or application.

Returning a Value

- Add a `RETURN` clause with the data type in the header of the function.
- Include one `RETURN` statement in the executable section.

Multiple `RETURN` statements are allowed in a function (usually within an `IF` statement). Only one `RETURN` statement is executed, because after the value is returned, processing of the block ceases.



Stored Function: Example

- Create the function:

```
CREATE OR REPLACE FUNCTION get_sal
 (id employees.employee_id%TYPE) RETURN NUMBER IS
  sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
  INTO   sal
  FROM   employees
  WHERE  employee_id = id;
  RETURN sal;
END get_sal;
/
```

- Invoke the function as an expression or as a parameter value:

```
EXECUTE dbms_output.put_line(get_sal(100))
```

2-6

Stored Function: Example

The `get_sal` function is created with a single input parameter and returns the salary as a number. Execute the command as shown, or save it in a script file and run the script to create the `get_sal` function.

The `get_sal` function follows a common programming practice of using a single `RETURN` statement that returns a value assigned to a local variable. If your function has an exception section, then it may also contain a `RETURN` statement.

Invoke a function as part of a PL/SQL expression because the function will return a value to the calling environment. The second code box uses the *i*SQL*Plus `EXECUTE` command to call the `DBMS_OUTPUT.PUT_LINE` procedure whose argument is the return value from the function `get_sal`. In this case, `get_sal` is invoked first to calculate the salary of the employee with ID 100. The salary value returned is supplied as the value of the `DBMS_OUTPUT.PUT_LINE` parameter, which displays the result (if you have executed a `SET SERVEROUTPUT ON`).

Note: A function must always return a value. The example does not return a value if a row is not found for a given `id`. Ideally, create an exception handler to return a value as well.

Ways to Execute Functions

- Invoke as part of a PL/SQL expression
  - Using a host variable to obtain the result:

```
VARIABLE salary NUMBER
EXECUTE :salary := get_sal(100)
```

  - Using a local variable to obtain the result:

```
DECLARE sal employees.salary%type;
BEGIN
  sal := get_sal(100); ...
END;
```

- Use as a parameter to another subprogram

```
EXECUTE dbms_output.put_line(get_sal(100))
```

- Use in a SQL statement (subject to restrictions)

```
SELECT job_id, get_sal(employee_id) FROM employees;
```

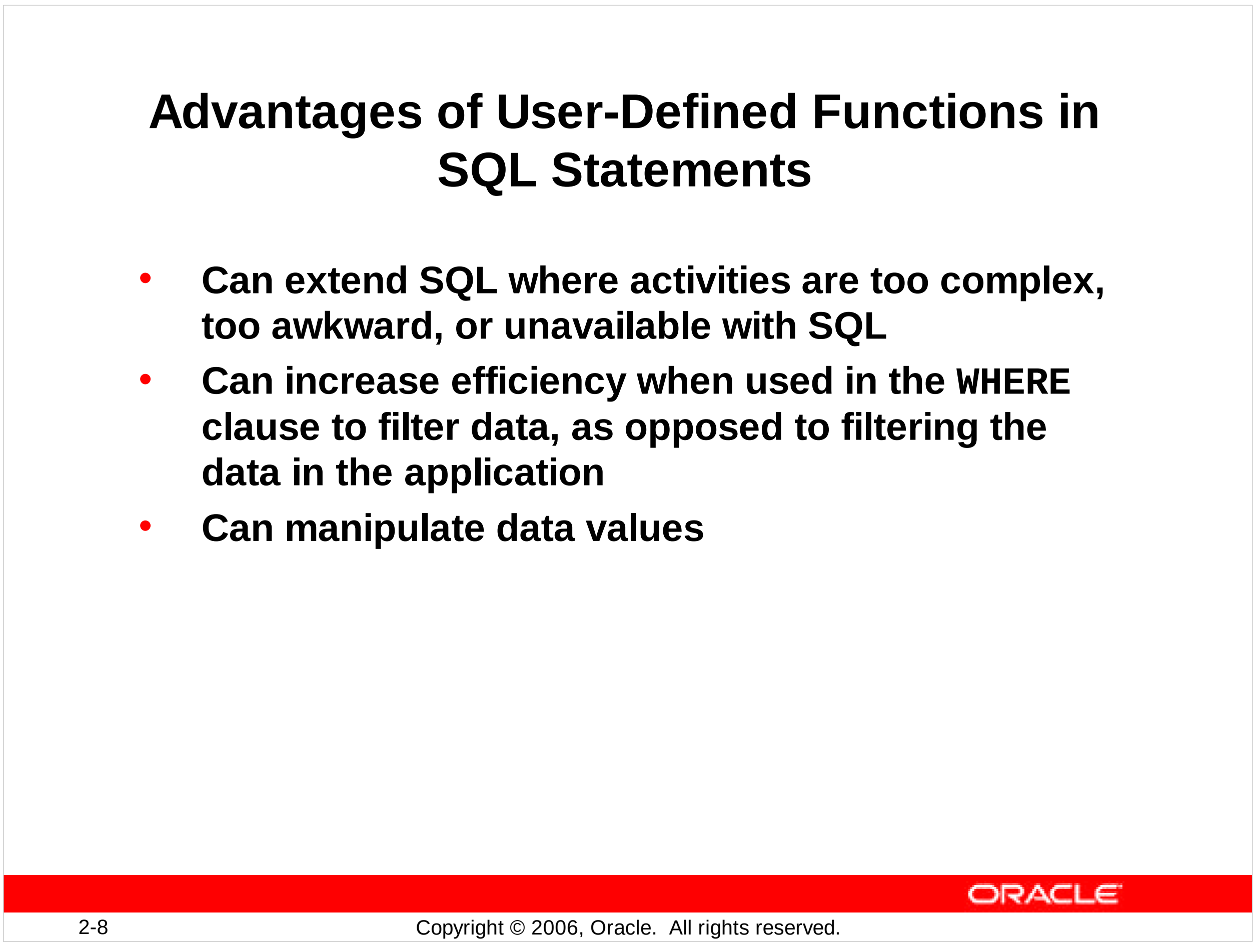
ORACLE

Ways to Execute Functions

If functions are designed thoughtfully, they can be powerful constructs. Functions can be invoked in the following ways:

- As part of PL/SQL expressions: You can use host or local variables to hold the returned value from a function. The first example in the slide uses a host variable and the second example uses a local variable in an anonymous block.
- As a parameter to another subprogram: The third example in the slide demonstrates this usage. The `get_sal` function with all its arguments is nested in the parameter required by the `DBMS_OUTPUT.PUT_LINE` procedure. This comes from the concept of nesting functions as discussed in the course titled *Oracle Database 10g: SQL Fundamentals I*.
- As an expression in a SQL statement: The last example shows how a function can be used as a single-row function in a SQL statement.

Note: The benefits and restrictions that apply to functions when used in a SQL statement are discussed in the next few pages.



Advantages of User-Defined Functions in SQL Statements

- Can extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application
- Can manipulate data values

Advantages of User-Defined Functions in SQL Statements

SQL statements can reference PL/SQL user-defined functions anywhere a SQL expression is allowed. For example, a user-defined function can be used anywhere that a built-in SQL function, such as `UPPER()`, can be placed.

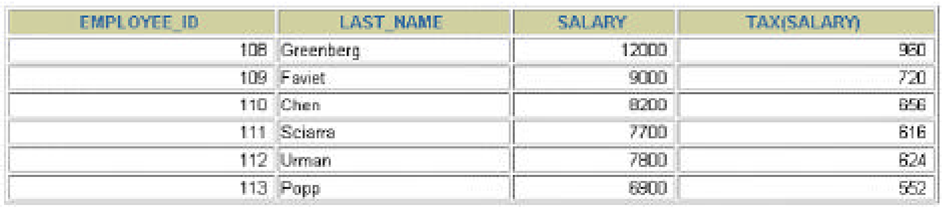
Advantages

- Permits calculations that are too complex, awkward, or unavailable with SQL
- Increases data independence by processing complex data analysis within the Oracle server, rather than by retrieving the data into an application
- Increases efficiency of queries by performing functions in the query rather than in the application
- Manipulates new types of data (for example, latitude and longitude) by encoding character strings and using functions to operate on the strings

Function in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(value IN NUMBER)
 RETURN NUMBER IS
BEGIN
   RETURN (value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

```
Function created.
```



EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

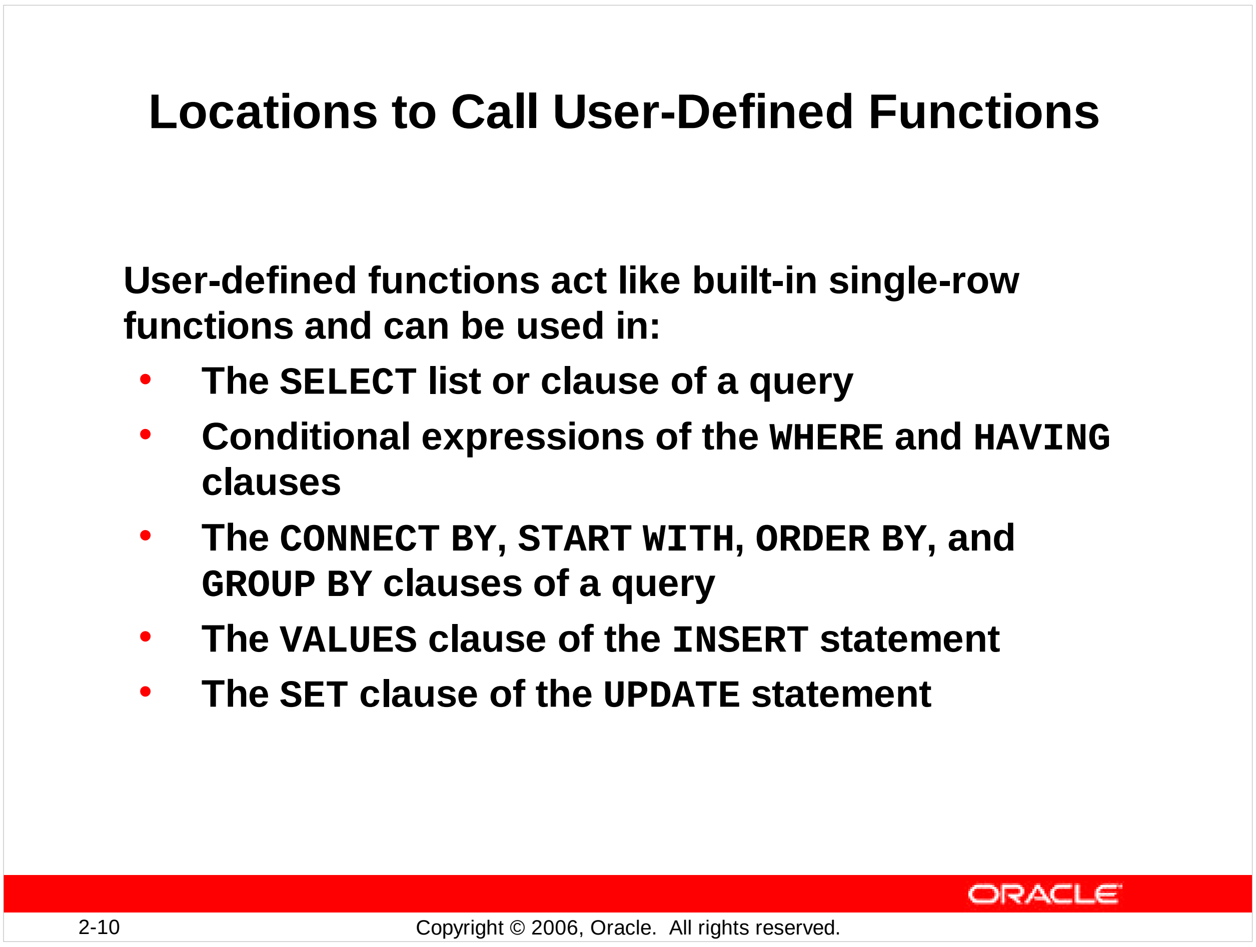
```
6 rows selected.
```

ORACLE

Function in SQL Expressions: Example

The example in the slide shows how to create a `tax` function to calculate income tax. The function accepts a `NUMBER` parameter and returns the calculated income tax based on a simple flat tax rate of 8%.

In *i*SQL*Plus, the `tax` function is invoked as an expression in the `SELECT` clause along with the employee ID, last name, and salary for employees in a department with ID `100`. The return result from the `tax` function is displayed with the regular output from the query.



Locations to Call User-Defined Functions

User-defined functions act like built-in single-row functions and can be used in:

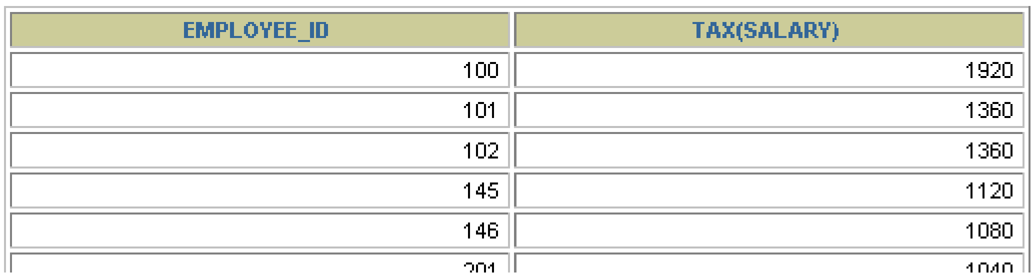
- The `SELECT` list or clause of a query
- Conditional expressions of the `WHERE` and `HAVING` clauses
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

Locations to Call User-Defined Functions

A PL/SQL user-defined function can be called from any SQL expression where a built-in single-row function can be called.

Example:

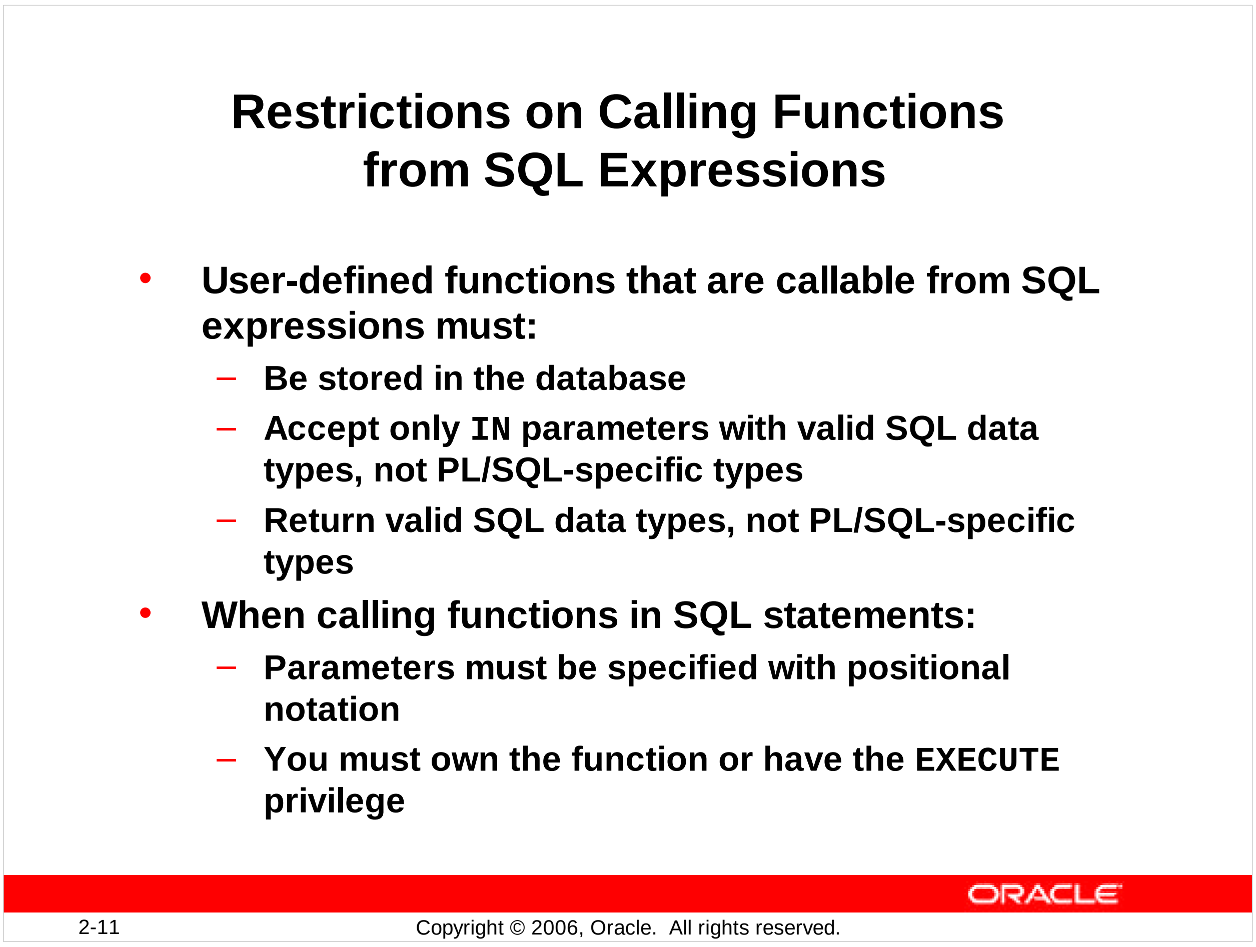
```
SELECT employee_id, tax(salary)
FROM   employees
WHERE  tax(salary) > (SELECT MAX(tax(salary))
                      FROM employees
```



EMPLOYEE_ID	TAX(SALARY)
100	1920
101	1360
102	1360
145	1120
146	1080
201	1040

...

10 rows selected.



Restrictions on Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  - Be stored in the database
  - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
  - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
  - Parameters must be specified with positional notation
  - You must own the function or have the `EXECUTE` privilege

2-11

Restrictions on Calling Functions from SQL Expressions

The user-defined PL/SQL functions that are callable from SQL expressions must meet the following requirements:

- The function must be stored in the database.
- The function parameters must be input only and valid SQL data types.
- The functions must return data types that are valid SQL data types. They cannot be PL/SQL-specific data types such as `BOOLEAN`, `RECORD`, or `TABLE`. The same restriction applies to the parameters of the function.

The following restrictions apply when calling a function in a SQL statement:

- Parameters must use positional notation. Named notation is not supported.
- You must own or have the `EXECUTE` privilege on the function.

Other restrictions on a user-defined function include the following:

- It cannot be called from the `CHECK` constraint clause of a `CREATE TABLE` or `ALTER TABLE` statement.
- It cannot be used to specify a default value for a column.

Controlling Side Effects When Calling Functions from SQL Expressions

Functions called from:

- **A `SELECT` statement cannot contain DML statements**
- **An `UPDATE` or `DELETE` statement on a table T cannot query or contain DML on the same table T**
- **SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)**

Note: Calls to subprograms that break these restrictions are also not allowed in the function.

Controlling Side Effects When Calling Functions from SQL Expressions

To execute a SQL statement that calls a stored function, the Oracle server must know whether the function is free of specific side effects. The side effects are unacceptable changes to database tables.

Additional restrictions apply when a function is called in expressions of SQL statements:

- When a function is called from a `SELECT` statement or a parallel `UPDATE` or `DELETE` statement, the function cannot modify database tables.
- When a function is called from an `UPDATE` or `DELETE` statement, the function cannot query or modify database tables modified by that statement.
- When a function is called from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute directly or indirectly through another subprogram or SQL transaction control statements such as:
  - A `COMMIT` or `ROLLBACK` statement
  - A session control statement (such as `SET ROLE`)
  - A system control statement (such as `ALTER SYSTEM`)
  - Any DDL statements (such as `CREATE`) because they are followed

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(sal NUMBER)
   RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name,
                 email, hire_date, job_id, salary)
  VALUES(1, 'Frost', 'jfrost@company.com',
         SYSDATE, 'SA_MAN', sal);
  RETURN (sal + 100);
END;
```

```
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170;
```

```
UPDATE employees SET salary = dml_call_sql(2000)
                *
ERROR at line 1:
ORA-04091: table PLSQL.EMPLOYEES is mutating,
trigger/function may not see it
ORA-06512: at "PLSQL.DML_CALL_SQL", line 4
```

Restrictions on Calling Functions from SQL: Example

The `dml_call_sql` function in the slide contains an `INSERT` statement that inserts a new record into the `EMPLOYEES` table and returns the input salary value incremented by 100. This function is invoked in the `UPDATE` statement that modifies the salary of employee 170 to the amount returned from the function. The `UPDATE` statement fails with an error indicating that the table is mutating (that is, changes are already in progress in the same table). In the following example, the `query_call_sql` function queries the `SALARY` column of the `EMPLOYEES` table:

```
CREATE OR REPLACE FUNCTION query_call_sql(a NUMBER)
  RETURN NUMBER IS
  s NUMBER;
 BEGIN
  SELECT salary INTO s FROM employees
  WHERE employee_id = 170;
  RETURN (s + a);
 END;
```

When invoked from the following `UPDATE` statement, it returns the error

Removing Functions

Removing a stored function:

- You can drop a stored function by using the following syntax:

```
DROP FUNCTION function_name
```

Example:

```
DROP FUNCTION get_sal;
```

- All the privileges that are granted on a function are revoked when the function is dropped.
- The `CREATE OR REPLACE` syntax is equivalent to dropping a function and re-creating it. Privileges granted on the function remain the same when this syntax is used.

Removing Functions

When a stored function is no longer required, you can use a SQL statement in *i*SQL*Plus to drop it. To remove a stored function by using *i*SQL*Plus, execute the `DROP FUNCTION` SQL command.

`CREATE OR REPLACE` Versus `DROP` and `CREATE`

The `REPLACE` clause in the `CREATE OR REPLACE` syntax is equivalent to dropping a function and re-creating it. When you use the `CREATE OR REPLACE` syntax, the privileges granted on this object to other users remain the same. When you `DROP` a function and then re-create it, all the privileges granted on this function are automatically revoked.

Viewing Functions in the Data Dictionary

Information for PL/SQL functions is stored in the following Oracle data dictionary views:

- **You can view source code in the `USER_SOURCE` table for subprograms that you own, or the `ALL_SOURCE` table for functions owned by others who have granted you the `EXECUTE` privilege.**

```
SELECT text
FROM   user_source
WHERE  type = 'FUNCTION'
ORDER BY line;
```

- **You can view the names of functions by using `USER_OBJECTS`.**

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'FUNCTION';
```

Viewing Functions in the Data Dictionary

The source code for PL/SQL functions is stored in the data dictionary tables. The source code is accessible for PL/SQL functions that are successfully or unsuccessfully compiled. To view the PL/SQL function code stored in the data dictionary, execute a `SELECT` statement on the following tables where the `TYPE` column value is `FUNCTION`:

- The `USER_SOURCE` table to display the PL/SQL code that you own
- The `ALL_SOURCE` table to display the PL/SQL code to which you have been granted the `EXECUTE` right by the owner of that subprogram code

The first query example shows how to display the source code for all the functions in your schema. The second query, which uses the `USER_OBJECTS` data dictionary view, lists the names of all functions that you own.

Procedures Versus Functions

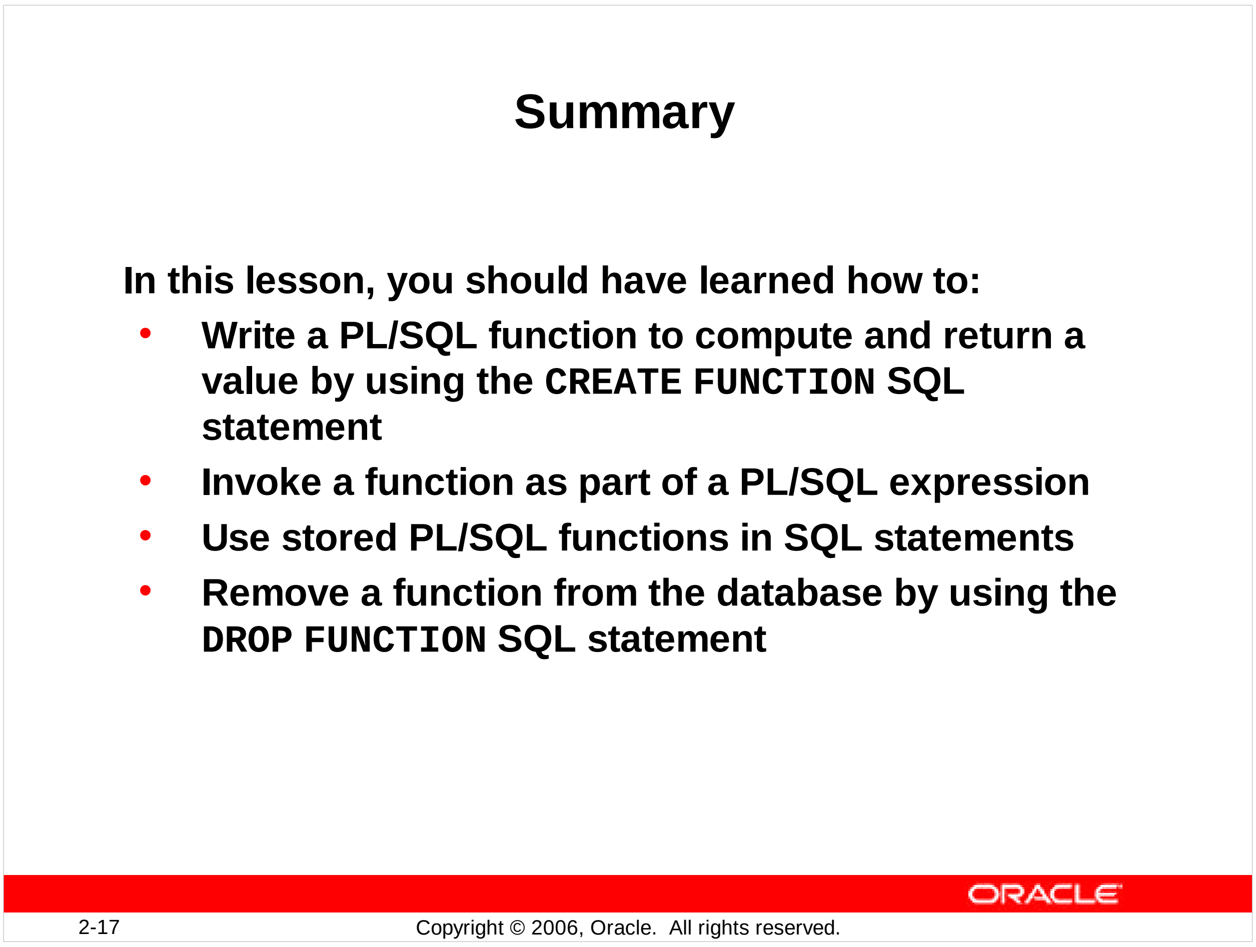
Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain `RETURN` clause in the header	Must contain a `RETURN` clause in the header
Can return values (if any) in output parameters	Must return a single value
Can contain a `RETURN` statement without a value	Must contain at least one `RETURN` statement

How Procedures and Functions Differ

You create a procedure to store a series of actions for later execution. A procedure can contain zero or more parameters that can be transferred to and from the calling environment, but a procedure does not have to return a value. A procedure can call a function to assist with its actions.

Note: A procedure containing a single `OUT` parameter would be better rewritten as a function returning the value.

You create a function when you want to compute a value that must be returned to the calling environment. A function can contain zero or more parameters that are transferred from the calling environment. Functions typically return only a single value, and the value is returned through a `RETURN` statement. The functions used in SQL statements should not use `OUT` or `IN OUT` mode parameters. Although a function using output parameters can be used in a PL/SQL procedure or block, it cannot be used in SQL statements.



Summary

In this lesson, you should have learned how to:

- Write a PL/SQL function to compute and return a value by using the `CREATE FUNCTION` SQL statement
- Invoke a function as part of a PL/SQL expression
- Use stored PL/SQL functions in SQL statements
- Remove a function from the database by using the `DROP FUNCTION` SQL statement

Summary

A function is a named PL/SQL block that must return a value. Generally, you create a function to compute and return a value, and you create a procedure to perform an action.

A function can be created or dropped.

A function is invoked as a part of an expression.

Practice 2: Overview

This practice covers the following topics:

- Creating stored functions:
  - To query a database table and return specific values
  - To be used in a SQL statement
  - To insert a new row, with specified parameter values, into a database table
  - Using default parameter values
- Invoking a stored function from a SQL statement
- Invoking a stored function from a stored procedure

Practice 2: Overview

If you encounter compilation errors when using *i*SQL*Plus, use the `SHOW ERRORS` command.

If you correct any compilation errors in *i*SQL*Plus, do so in the original script file, not in the buffer, and then rerun the new version of the file. This saves a new version of the program unit to the data dictionary.

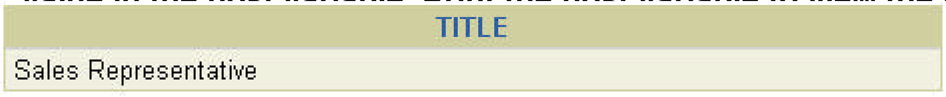
Note: It is recommended to use *i*SQL*Plus for this practice.

Practice 2

1. Create and invoke the `GET_JOB` function to return a job title.

   a. Create and compile a function called `GET_JOB` to return a job title.

   b. Create a `VARCHAR2` host variable called `TITLE`, allowing a length of 35 characters. Invoke the function with `SA_REP` job ID to return the value in the host variable. Print the host variable to view the result.

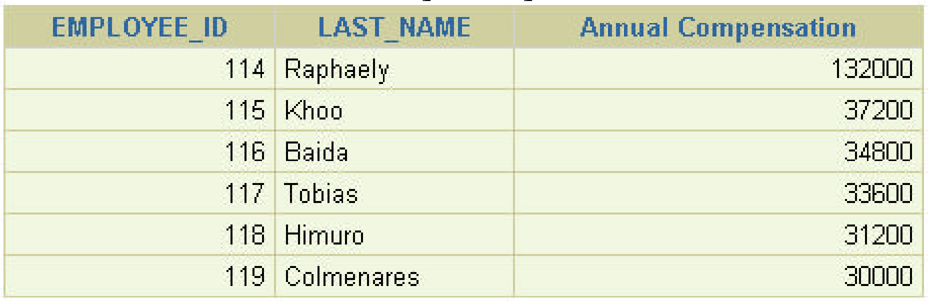


TITLE
Sales Representative

2. Create a function called `GET_ANNUAL_COMP` to return the annual salary computed from an employee's monthly salary and commission passed as parameters.

   a. Develop and store the `GET_ANNUAL_COMP` function, accepting parameter values for monthly salary and commission. Either or both v n a r lculate t

   b. U S table f



EMPLOYEE_ID	LAST_NAME	Annual Compensation
114	Raphaely	132000
115	Khoo	37200
116	Baida	34800
117	Tobias	33600
118	Himuro	31200
119	Colmenares	30000

6 rows selected.

3. Create a procedure, `ADD_EMPLOYEE`, to insert a new employee into the `EMPLOYEES` table. The procedure should call a `VALID_DEPTID` function to check whether the department ID specified for the new employee exists in the `DEPARTMENTS` table.