9 Creating Procedures and Using Parameters

✅Create a procedure with parameters

Subprogram parameters: 8.7 - 8.7.5

Formal and actual subprogram parameters

- If you want a subprogram to have parameters, declare formal parameters in the subprogram heading
- In each formal parameter declaration, specify the name and data type of the parameter, and (optionally) its mode and default value
- In the execution part of the subprogram, reference the formal parameters by their names
- When invoking the subprogram, specify the actual parameters whose values are to be assigned to the formal parameters
- Corresponding actual and formal parameters must have compatible data types
- > Note: You can declare a formal parameter of a constrained subtype, like this: `DECLARE SUBTYPE n1 IS NUMBER(1); SUBTYPE v1 IS VARCHAR2(1); PROCEDURE p (n n1, v v1) IS ...`

  > But you cannot include a constraint in a formal parameter declaration, like this: `DECLARE PROCEDURE p (n NUMBER(1), v VARCHAR2(1)) IS ...`
- > Tip: To avoid confusion, use different names for formal and actual parameters
- > Note:
  >
  > - Actual parameters (including default values of formal parameters) can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.
  > - You cannot use LOB parameters in a server-to-server remote procedure call (RPC)

Formal parameters of constrained subtypes If the data type of a formal parameter is a constrained subtype, then:

- If the subtype has the `NOT NULL` constraint, then the actual parameter inherits it
- If the subtype has the base type `VARCHAR2`, then the actual parameter does not inherit the size of the subtype
- If the subtype has a numeric base type, then the actual parameter inherits the range of the subtype, but not the precision or scale
- > Note: In a function, the clause `RETURN` datatype declares a hidden formal parameter and the statement `RETURN` value specifies the corresponding actual parameter. Therefore, if datatype is a constrained data type, then the preceding rules apply to value
- PL/SQL has many predefined data types that are constrained subtypes of other data types. For example, `INTEGER` is a constrained subtype of `NUMBER`
- example: take a function that has both an `INTEGER` formal parameter and an `INTEGER` return type:
  - The anonymous block invokes the function with an actual parameter that is not an integer (but a floating point)

- Because the actual parameter inherits the range but not the precision and scale of `INTEGER`, and the actual parameter is in the `INTEGER` range, the invocation succeeds.
- For the same reason, the `RETURN` statement succeeds in returning the noninteger value

Subprogram parameter passing methods

The PL/SQL compiler has two ways of passing an actual parameter to a subprogram:

- **By reference**
  - The compiler passes the subprogram a pointer to the actual parameter. The actual and formal parameters refer to the same memory location
- **By value**
  - The compiler assigns the value of the actual parameter to the corresponding formal parameter. The actual and formal parameters refer to different memory locations
  - If necessary, the compiler implicitly converts the data type of the actual parameter to the data type of the formal parameter
  - > Tip: Avoid implicit data conversion, in either of these ways:
    >
    > - Declare the variables that you intend to use as actual parameters with the same data types as their corresponding formal parameters
    > - Explicitly convert actual parameters to the data types of their corresponding formal parameters, using the SQL conversion functions
- The method by which the compiler passes a specific actual parameter depends on its mode (subprogram parameter modes)

Subprogram parameter modes

The **mode** of a formal parameter determines its behavior, these are the parameter modes and their characteristics:

- IN

  - **is default?**: Default mode
  - **role**: Passes a value to the subprogram
  - **formal parameter**: Formal parameter acts like a constant: When the subprogram begins, its value is that of either its actual parameter or default value, and the subprogram cannot change this value
  - **actual parameter**: Actual parameter can be a constant, initialized variable, literal, or expression
  - **passed by reference?**: Actual parameter is passed by reference

- OUT

  - **is default?**: Must be specified
  - **role**: Returns a value to the invoker
  - **formal parameter**: Formal parameter is initialized to the default value of its type. The default value of the type is `NULL` except for a record type with a non-`NULL` default value. When the subprogram begins, the formal parameter has its initial value regardless of the value of its actual parameter. Oracle recommends that the subprogram assign a value to the formal parameter

  - **actual parameter**: If the default value of the formal parameter type is `NULL`, then the actual parameter must be a variable whose data type is not defined as `NOT NULL`
  - **passed by reference?**: By default, actual parameter is passed by value; if you specify `NOCOPY`, it might be passed by reference

- IN OUT

  - **is default?**: Must be specified
  - **role**: Passes an initial value to the subprogram and returns an updated value to the invoker
  - **formal parameter**: Formal parameter acts like an initialized variable: When the subprogram begins, its value is that of its actual parameter. Oracle recommends that the subprogram update its value
  - **actual parameter**: Actual parameter must be a variable (typically, it is a string buffer or numeric accumulator)
  - **passed by reference?**: By default, actual parameter is passed by value (in both directions); if you specify `NOCOPY`, it might be passed by reference

- > Tip: Do not use OUT and IN OUT for function parameters. Ideally, a function takes zero or more parameters and returns a single value. A function with IN OUT parameters returns multiple values and has side effects

- Regardless of how an OUT or IN OUT parameter is passed:

  - If the subprogram exits successfully, then the value of the actual parameter is the final value assigned to the formal parameter. (The formal parameter is assigned at least one value—the initial value)
  - If the subprogram ends with an exception, then the value of the actual parameter is undefined
  - Formal OUT and IN OUT parameters can be returned in any order

- When an OUT or IN OUT parameter is passed by reference, the actual and formal parameters refer to the same memory location. Therefore, if the subprogram changes the value of the formal parameter, the change shows immediately in the actual parameter

Subprogram parameter aliasing

- Aliasing is having two different names for the same memory location. If a stored item is visible by more than one path, and you can change the item by one path, then you can see the change by all paths
- Subprogram parameter aliasing always occurs when the compiler passes an actual parameter by reference, and can also occur when a subprogram has cursor variable parameters

Subprogram parameter aliasing with parameters passed by reference

- When the compiler passes an actual parameter by reference, the actual and formal parameters refer to the same memory location. Therefore, if the subprogram changes the value of the formal parameter, the change shows immediately in the actual parameter
- The compiler always passes `IN` parameters by reference, but the resulting aliasing cannot cause problems, because subprograms cannot assign values to `IN` parameters
- The compiler *might* pass an `OUT` or `IN OUT` parameter by reference, if you specify `NOCOPY` for that parameter

- `NOCOPY` is only a hint—each time the subprogram is invoked, the compiler decides, silently, whether to obey or ignore `NOCOPY`. Therefore, aliasing can occur for one invocation but not another, making subprogram results indeterminate. For example:
  - If the actual parameter is a global variable, then an assignment to the formal parameter might show in the global parameter
  - If the same variable is the actual parameter for two formal parameters, then an assignment to either formal parameter might show immediately in both formal parameters
  - If the actual parameter is a package variable, then an assignment to either the formal parameter or the package variable might show immediately in both the formal parameter and the package variable
  - If the subprogram is exited with an unhandled exception, then an assignment to the formal parameter might show in the actual parameter

Subprogram parameter aliasing with cursor variable parameters

- Cursor variable parameters are pointers. Therefore, if a subprogram assigns one cursor variable parameter to another, they refer to the same memory location
- This aliasing can have unintended results

Default values for IN subprogram parameters

- When you declare a formal `IN` parameter, you can specify a default value for it. A formal parameter with a default value is called an **optional parameter**, because its corresponding actual parameter is optional in a subprogram invocation
- If the actual parameter is omitted, then the invocation assigns the default value to the formal parameter
- A formal parameter with no default value is called a **required parameter**, because its corresponding actual parameter is required in a subprogram invocation
- Omitting an actual parameter does not make the value of the corresponding formal parameter `NULL`
- To make the value of a formal parameter `NULL`, specify `NULL` as either the default value or the actual parameter
- The default value of a formal parameter can be any expression whose value can be assigned to the parameter; that is, the value and parameter must have compatible data types
- If a subprogram invocation specifies an actual parameter for the formal parameter, then that invocation does not evaluate the default value

✅Use named notation

Positional, named and mixed notation: 8.7.6

When invoking a subprogram, you can specify the actual parameters using either positional, named, or mixed notation

- **Positional**
  - syntax:
    - Specify the actual parameters in the same order as the formal parameters are declared
  - optional parameters:
    - You can omit trailing optional parameters
  - advantages:

        - N/A
    - disadvantages:
        - Specifying actual parameters in the wrong order can cause problems that are hard to detect, especially if the actual parameters are literals
        - Subprogram invocations must change if the formal parameter list changes, unless the list only acquires new trailing optional parameters
        - Reduced code clarity and maintainability. Not recommended if the subprogram has a large number of parameters

- **Named**

    - syntax:
        - Specify the actual parameters in any order, using this syntax: `formal => actual`
        - formal is the name of the formal parameter and actual is the actual parameter
    - optional parameters:
        - You can omit any optional parameters
    - advantages:
        - There is no wrong order for specifying actual parameters
        - Subprogram invocations must change only if the formal parameter list acquires new required parameters
        - Recommended when you invoke a subprogram defined or maintained by someone else
    - disadvantages:
        - N/A

- **Mixed**

    - syntax:
        - Start with positional notation, then use named notation for the remaining parameters
    - optional parameters:
        - In the positional notation, you can omit trailing optional parameters; in the named notation, you can omit any optional parameters
    - advantages:
        - Convenient when you invoke a subprogram that has required parameters followed by optional parameters, and you must specify only a few of the optional parameters
    - disadvantages:
        - In the positional notation, the wrong order can cause problems that are hard to detect, especially if the actual parameters are literals
        - Changes to the formal parameter list might require changes in the positional notation

- Examples:

```
CREATE OR REPLACE FUNCTION compute_bonus (
    emp_id NUMBER,
    salary NUMBER,
    bonus NUMBER DEFAULT 0.1,
    elligible VARCHAR2 DEFAULT 'NO'
) RETURN NUMBER IS
BEGIN
```

```
  ...
  RETURN ...
END;
/
DECLARE
  i NUMBER;
  emp_num NUMBER := 77;
  sal NUMBER := 1500
BEGIN
  i := compute_bonus(emp_num, sal, 0.2); -- positional notation
  i := compute_bonus(elligible => 'YES', emp_id => emp_num, salary => sal);
-- named notation
  i := compute_bonus(emp_num, sal, elligible => 'YES'); -- mixed notation
END;
/
SELECT compute_bonus(77, salary => 1500) FROM DUAL; -- mixed notation
SELECT compute_bonus(salary => 1500, emp_id => 77, bonus => 0.3) FROM DUAL;
-- named notation
SELECT compute_bonus(77, 1500, 0.15, 'YES') FROM DUAL; -- positional
notation
```

✅Work with procedures (create, invoke and remove procedures)

Forward declaration: 8.6 Procedure declaration and definition: 13.50 CREATE PROCEDURE ALTER PROCEDURE DROP PROCEDURE

Forward declaration

- If nested subprograms in the same PL/SQL block invoke each other, then one requires a forward declaration, because a subprogram must be declared before it can be invoked
- A forward declaration declares a nested subprogram but does not define it. You must define it later in the same block
- The forward declaration and the definition must have the same subprogram heading
- Example:

```
DECLARE
  -- declare proc1 (forward declaration):
  PROCEDURE proc1(number1 NUMBER);

  -- declare and define proc2:
  PROCEDURE proc2(number2 NUMBER) IS
  BEGIN
    proc1(number2);
  END;

  -- define proc 1:
```

```
  PROCEDURE proc1(number1 NUMBER) IS
  BEGIN
    proc2 (number1);
  END;

BEGIN
  NULL;
END;
/
```

Procedure declaration and definition

- Before invoking a procedure, you must declare and define it. You can either declare it first (with procedure_declaration) and then define it later in the same block, subprogram, or package (with procedure_definition) or declare and define it at the same time (with procedure_definition)
- A procedure is a subprogram that performs a specific action. A procedure invocation (or call) is a statement
- A procedure declaration is also called a procedure specification or procedure spec
- procedure_declaration: `PROCEDURE procedure [(parameter_declaration [,...])] [accessible_by] [default_collation] [invoker_rights]`
  - creates a procedure but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration
  - Each procedure property can appear only once in the procedure declaration. The properties can appear in any order
- procedure_definition: `procedure_declaration {IS | AS} {[declare_section] body | call_spec}`
  - Either defines a procedure that was declared earlier or both declares and defines a procedure
  - declare_section declares items that are local to the procedure, can be referenced in body, and cease to exist when the procedure completes execution
  - body is the required executable part and optional exception-handling part of the procedure

CREATE PROCEDURE statement

- The `CREATE PROCEDURE` statement creates or replaces a standalone procedure or a call specification

- A standalone procedure that you create with the `CREATE PROCEDURE` statement differs from a procedure that you declare and define in a PL/SQL block or package

- A call specification declares a Java method or a C language procedure so that it can be called from PL/SQL. You can also use the SQL CALL statement to invoke such a method or subprogram

- The call specification tells the database which Java method, or which named procedure in which shared library, to invoke when an invocation is made. It also tells the database what type conversions to make for the arguments and return value

- Prerequisites

  - To create or replace a standalone procedure in your schema, you must have the `CREATE PROCEDURE` system privilege

  - To create or replace a standalone procedure in another user's schema, you must have the `CREATE ANY PROCEDURE` system privilege
  - To invoke a call specification, you may need additional privileges, for example, the `EXECUTE` object privilege on the C library for a C call specification
  - To embed a `CREATE PROCEDURE` statement inside an Oracle precompiler program, you must terminate the statement with the keyword `END-EXEC` followed by the embedded SQL statement terminator for the specific language

- syntax:

```
CREATE [OR REPLACE] [EDITIONABLE | NONEDITIONABLE] PROCEDURE [schema.]
procedure_name
[(parameter_declaration [,...])] [sharing_clause] [default_collation,
invokers_rights, accessible_by]
{IS | AS} {[declare_section] body | call_spec} ;
```

  - `OR REPLACE`:
    - Re-creates the procedure if it exists, and recompiles it
    - Users who were granted privileges on the procedure before it was redefined can still access the procedure without being regranted the privileges
    - If any function-based indexes depend on the procedure, then the database marks the indexes `DISABLED`
  - `[ EDITIONABLE | NONEDITIONABLE ]`:
    - Specifies whether the procedure is an editioned or noneditioned object if editioning is enabled for the schema object type `PROCEDURE` in schema
    - Default: `EDITIONABLE`

ALTER PROCEDURE statement

- The `ALTER PROCEDURE` statement explicitly recompiles a standalone procedure
- Explicit recompilation eliminates the need for implicit runtime recompilation and prevents associated runtime compilation errors and performance overhead
- To recompile a procedure that is part of a package, recompile the entire package using the `ALTER PACKAGE` statement
- > Note: This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a standalone procedure, use the `CREATE PROCEDURE` statement with the `OR REPLACE` clause
- Prerequisites
  - If the procedure is in the SYS schema, you must be connected as SYSDBA
  - Otherwise, the procedure must be in your schema or you must have ALTER ANY PROCEDURE system privilege
- syntax:

```
ALTER PROCEDURE [schema.] procedure_name {procedure_compile_clause |
{EDITIONABLE | NONEDITIONABLE}};
```

  - procedure_compile_clause: `COMPILE [DEBUG] [compiler_parameters_clause] [REUSE SETTINGS]`

- Example:
  - To explicitly recompile the procedure remove_emp owned by the user hr, issue this statement: `ALTER PROCEDURE hr.remove_emp COMPILE;`
  - If the database encounters no compilation errors while recompiling remove_emp, then remove_emp becomes valid. The database can subsequently run it without recompiling it at run time
  - If recompiling remove_emp results in compilation errors, then the database returns an error and remove_emp remains invalid
  - The database also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that invoke remove_emp
  - If you subsequently reference one of these objects without first explicitly recompiling it, then the database recompiles it implicitly at run time

DROP PROCEDURE

- The `DROP PROCEDURE` statement drops a standalone procedure from the database
- > Note: Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the `DROP PACKAGE` statement, or redefine the package without the procedure using the `CREATE PACKAGE` statement with the `OR REPLACE` clause

- Prerequisites
  - The procedure must be in your schema or you must have the `DROP ANY PROCEDURE` system privilege
- syntax: `DROP PROCEDURE [schema.] procedure ;`
- When you drop a procedure, the database invalidates any local objects that depend upon the dropped procedure
- If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error message if you have not re-created the dropped procedure

✅Handle exceptions in procedures and display a procedure's information

DESCRIBE *_OBJECTS views *_PROCEDURES views *_ARGUMENTS views *_SOURCE views *_ERRORS views DBA_OBJECT_SIZE view

DESCRIBE

- syntax: `DESC[RIBE] {[schema.]object[@db_link]}` lists the column definitions for the specified table, view or synonym, or the specifications for the specified function or procedure

- The description for functions and procedures contains the following information:
    - the type of PL/SQL object (function or procedure)
    - the name of the function or procedure
    - the type of value returned (for functions)
    - the argument names, types, whether input or output, and default values, if any
    - the `ENCRYPT` keyword to indicate whether or not data in a column is encrypted
- Example: describe a procedure called customer_lookup `DESCRIBE customer_lookup` output:

```
PROCEDURE customer_lookup
Argument Name           Type     In/Out   Default?
----------------------  -------- -------- ---------
CUST_ID                 NUMBER   IN
CUST_NAME               VARCHAR2 OUT
```

    - can also describe packages. Each function or procedure in the package specification will then be described separately

*_OBJECTS views

- `ALL_OBJECTS` describes all objects accessible to the current user
- related: `DBA_OBJECTS, USER_OBJECTS`
- notable columns:
    - OWNER
    - OBJECT_NAME
    - OBJECT_ID: dictionary object number
    - OBJECT_TYPE: table, index, package, package_body, function, etc.
    - CREATED: date for creation
    - LAST_DDL_TIME: date for last modification of the object and dependent objects resulting from DDL
    - TIMESTAMP: timestamp for specification of the object
    - STATUS: valid, invalid, N/A

*_PROCEDURES views

- `ALL_PROCEDURES` lists all functions and procedures that are accessible to the current user, along with associated properties
- For example, `ALL_PROCEDURES` indicates whether or not a function is pipelined, parallel enabled or an aggregate function. If a function is pipelined or an aggregate function, the associated implementation type (if any) is also identified
- related: `DBA_PROCEDURES, USER_PROCEDURES`
- notable columns:
    - OBJECT_NAME
    - AGGREGATE: yes, no
    - PIPELINED: yes, no
    - PARALLEL: yes, no

  - DETERMINISTIC: yes, no
  - AUTHID: definer, current_user
  - RESULT_CACHE: yes, no

*_ARGUMENTS

- `ALL_ARGUMENTS` lists the arguments of the functions and procedures that are accessible to the current user
- Starting with Oracle Database 12c release 1 (12.1.0.2), this view omits procedures with no arguments. Prior to Oracle Database 12c release 1 (12.1.0.2), a procedure with no arguments was presented as a single row in this view
- Starting with Oracle Database 18c, this view displays only one row for an argument that is a composite type. Prior to Oracle Database 18c, this view displayed multiple rows for composite types
- related: DBA_ARGUMENTS, USER_ARGUMENTS
- notable columns:
  - OBJECT_NAME
  - PACKAGE_NAME
  - ARGUMENT_NAME
  - POSITION: 1 .. n for arg list position, 0 for function return val
  - SEQUENCE: sequential order of the argument, starts from 1, return type comes first
  - DATA_TYPE
  - DEFAULTED
  - IN_OUT: in, out, in/out
  - DATA_LENGTH
  - DATA_PRECISION

*_SOURCE views

- `ALL_SOURCE` describes the text source of the stored objects accessible to the current user
- related: `DBA_SOURCE, USER_SOURCE`
- notable columns:
  - OWNER
  - NAME
  - TYPE
  - LINE
  - TEXT

*_ERRORS

- `ALL_ERRORS` describes the current errors on the stored objects accessible to the current user
- related: `DBA_ERRORS, USER_ERRORS`
- notable columns:
  - NAME
  - TYPE: type of the object
  - LINE: line number at which error occurred
  - POSITION: position in line where error
  - TEXT: text of the error
  - ATTRIBUTE: indicates whether error or warning

- MESSAGE_NUMBER: numeric error number (without prefix)

DBA_OBJECT_SIZE

- `DBA_OBJECT_SIZ`E lists the sizes, in bytes, of various PL/SQL objects
- related: `USER_OBJECT_SIZE`
- notable columns:
  - NAME
  - TYPE
  - SOURCE_SIZE
  - PARSED_SIZE
  - CODE_SIZE
  - ERROR_SIZE