

# Semester 1 Final Review

# Semester 1 Final Review

**This slide set includes the following topics:**

- **Handling Exceptions**
- **Trapping Oracle Server Exceptions**
- **Trapping User-Defined Exceptions**
- **Recognizing the Scope of Variables and Exceptions**
- **Creating Procedures**
- **Using Parameters in Procedures**
- **Passing Parameters**
- **Creating Functions**
- **Using Functions in SQL Statements**
- **Reviewing the Data Dictionary**
- **Managing Procedures and Functions**
- **Reviewing Object Privileges**
- **Using Invoker's Rights**

# Handling Exceptions



## What Is an Exception?

An exception occurs when an error is discovered during the execution of a program that disrupts its normal operation.

## What Is an Exception Handler?

An exception handler is code that defines the recovery actions to be performed when exceptions are raised (that is, when errors occur).

When writing code, programmers need to anticipate the types of errors that can occur during execution of that code. They need to include exception handlers in their code to address these errors. In a sense, exception handlers allow programmers to "bulletproof" their code.

 **What Is an Exception Handler? (continued)**

What types of errors might programmers want to account for with exception errors?

- System errors (for example, memory)
- Data errors (for example, duplicate value in a unique index)
- User-action errors (for example, data entry error)

**Why Is Exception Handling Important?**

Exception handling protects the user from errors (frequent errors can frustrate the user and/or cause the user to quit the application). It also protects the database from errors (data can be lost or overwritten).

Major errors take a lot of system resources (if a mistake is made, correcting the mistake can be costly; users might frequently call the help desk for assistance with errors). Code is more readable because you can write error-handling routines in the same block in which the error occurred.



## Exceptions in PL/SQL

```
DECLARE
  v_country_name wf_countries.country_name%TYPE
                := 'Korea, South';
  v_elevation wf_countries.highest_elevation%TYPE;

BEGIN
  SELECT highest_elevation
         INTO v_elevation
         FROM wf_countries
         WHERE country_name = v_country_name;
END;
```

The code does not work as expected and the program terminates with the following error message.

```
ORA-01403: no data found
```

No data was found for Korea, South because the country name is actually stored as Republic of Korea.



## Handling Exceptions With PL/SQL

A block always terminates when PL/SQL raises an exception, but you can specify an exception handler to perform final actions before the block ends. The exception section begins with the keyword `EXCEPTION`.

```
DECLARE
  v_country_name wf_countries.country_name%TYPE
                := 'Korea, South';
  v_elevation wf_countries.highest_elevation%TYPE;
BEGIN
  SELECT highest_elevation INTO v_elevation
     FROM wf_countries WHERE country_name = v_country_name;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('Country name, ' || v_country_name || ',
      cannot be found. Re-enter the country name using the correct
      spelling. ');
END;
```

## Handling Exceptions With PL/SQL (continued)

When an exception is handled, the PL/SQL program does not terminate abruptly. When the exception is raised, the control shifts to the exception section and the handler in the exception section is executed. In the preceding example, the PL/SQL block terminates with normal, successful completion.

```
Country name, Korea, South,  
Cannot be found. Re-enter he country name using the correct spelling.  
  
Statement processed.
```

Only one handler is processed by PL/SQL before leaving the block.

## Handling Exceptions With PL/SQL (continued)

The following is another example of an exception being raised when the select statement in the block attempts to retrieve the last\_name of John.

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname FROM employees
  WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is : ' ||v_lname);
END;
```

An exception is raised because more than one John exists in the data.

```
ORA-01422: exact fetch returns more than requested number of rows
```



## Handling Exceptions With PL/SQL

The following code includes an exception handler for the predefined Oracle server error called `TOO_MANY_ROWS`.

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
     FROM employees WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John's last name is : ' || v_lname);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE ('Your select statement
    retrieved multiple rows. Consider using a
    cursor. ');
END;
```



## Trapping Exceptions

You can handle or "trap" any error by including a corresponding handler within the exception handling section of the PL/SQL block.

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

 **Trapping Exceptions (continued)**

Each handler consists of a `WHEN` clause, which specifies an exception name, followed by one or more statements to be executed when that exception is raised. You can include any number of handlers within an `EXCEPTION` section to handle specific exceptions. However, you cannot have multiple handlers for a single exception. The `WHEN OTHERS THEN` is an optional exception-handling clause that traps any exceptions that have not been explicitly handled. It must be the last exception handler that is defined.

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```



## Guidelines for Trapping Exceptions

- Add exception handlers whenever there is any possibility of an error occurring. Errors are especially likely during calculations, string manipulation, and SQL database operations.
- Handle named exceptions whenever possible, instead of using `OTHERS` in exception handlers. Learn the names and causes of the predefined exceptions.
- Test your code with different combinations of bad data to see what potential errors arise.
- Write out debugging information in your exception handlers.
- Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue. No matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.

# Trapping Oracle Server Exceptions



## Three Exception Types

Each of the three types is handled in a slightly different manner.

Exception	Description	Instructions for Handling
Predefined Oracle server error	One of approximately 20 errors that occur most often in PL/SQL code	You need not declare these exceptions. They are predefined by the Oracle server and are raised implicitly.
Non-predefined Oracle server error	Any other standard Oracle server error	Declare within the declarative section and allow the Oracle server to raise them implicitly.
User-defined error	A condition that the developer determines is abnormal	Declare within the declarative section, and raise explicitly.



## Handling Exceptions with PL/SQL

There are two ways to raise exceptions:

- Implicitly (automatically) by the Oracle server: An Oracle error occurs and the associated exception is raised automatically. For example, if the error `ORA-01403` occurs when no rows are retrieved from the database in a `SELECT` statement, then PL/SQL raises the exception `NO_DATA_FOUND`.
- Explicitly by the program: As a developer, depending on the business functionality your program is implementing, you might have to explicitly raise an exception. You raise an exception explicitly by issuing the `RAISE` statement within the block. The exception being raised can be either user-defined or predefined.

## Two Types of Oracle Server Error

When an Oracle server error occurs, the Oracle server automatically raises the associated exception, skips the rest of the executable section of the block, and looks for a handler in the exception section. There are two types of Oracle server errors:

- **Predefined Oracle server errors:** Each of these has a predefined name. For example, if the error `ORA-01403` occurs when no rows are retrieved from the database in a `SELECT` statement, then PL/SQL raises the predefined exception-name `NO_DATA_FOUND`.
- **Non-predefined Oracle server errors:** Each of these has a standard Oracle error number (`ORA-nnnnnn`) and error message, but not a predefined name. You declare your own names for these so that you can reference these names in the exception section.

## Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception handling routine.
- Sample predefined exceptions:
  - NO\_DATA\_FOUND
  - TOO\_MANY\_ROWS
  - INVALID\_CURSOR
  - ZERO\_DIVIDE
  - DUP\_VAL\_ON\_INDEX
- For a partial list of predefined exceptions, refer to `plsql_s06_102_predefined_errors.doc`. For a complete list of predefined exceptions, see the *PL/SQL User's Guide and Reference*.

## Trapping Predefined Oracle Server Errors (continued)

The following example uses the `TOO_MANY_ROWS` predefined Oracle Server error. Note that it is not declared in the `DECLARATION` section.

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
     FROM employees WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John's last name is : ' || v_lname);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE (' Your select statement
    retrieved multiple rows. Consider using a
    cursor. ');
END;
```

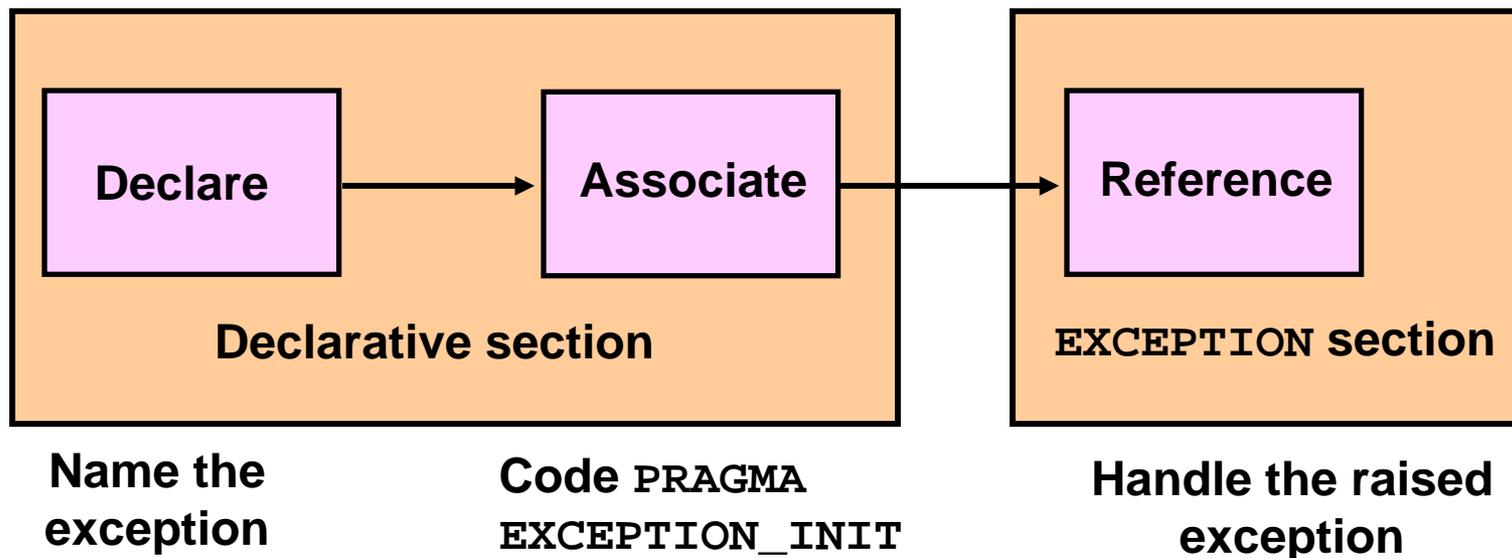
## Trapping Several Predefined Oracle Server Errors (continued)

The following example handles both the `TOO_MANY_ROWS` and `NO_DATA_FOUND` and also includes a `WHEN OTHERS` handler in case any other error occurs.

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
  FROM employees WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John's last name is : ' || v_lname);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE ('Select statement found multiple rows');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('Select statement found no rows');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('Another type of error occurred');
END;
```

## Trapping Non-Predefined Oracle Server Errors

Non-predefined exceptions are similar to predefined exceptions; however, they do not have predefined names in PL/SQL. They are standard Oracle Server errors and have ORA- error numbers. You can create your own names for them in the `DECLARE` section, and associate these names with ORA- error numbers using the `PRAGMA EXCEPTION_INIT` function.



## Trapping Non-Predefined Oracle Server Errors (continued)

- You can trap a non-predefined Oracle server error by declaring it first. The declared exception is raised implicitly. In PL/SQL, the `PRAGMA EXCEPTION_INIT` tells the compiler to associate an exception name with an Oracle error number.
- This allows you to refer to any internal exception by name and to write a specific handler for it.



## Non-Predefined Error

In the following example, the `INSERT` statement tries to insert the value `NULL` for the `department_name` column. It fails because the `department_name` column definition includes a not null constraint. There is no predefined error code for inserting nulls. The way to work around this problem is to create a non-predefined exception handler that traps the `ORA-1400` error.

```
BEGIN
  INSERT INTO departments
    (department_id, department_name) VALUES (280, NULL);
END;
```

```
ORA-01400: cannot insert NULL into ("USVA_TEST_SQL01_S01"."DEPARTMENTS"."DEPARTMENT_NAME")
```

 **Non-Predefined Error (continued)**

By modifying the previous code to include a non-predefined handler to trap Oracle server error number 1400, the error is trapped and handled.

```
DECLARE
  e_insert_excep EXCEPTION;
  PRAGMA EXCEPTION_INIT
    (e_insert_excep, -01400);
BEGIN
  INSERT INTO departments
    (department_id, department_name)
    VALUES (280, NULL);
EXCEPTION
  WHEN e_insert_excep
  THEN
    DBMS_OUTPUT.PUT_LINE('INSERT FAILED');
END;
```

1. Declare the name of the exception in the declarative section.
2. Use the Pragma to associate the name with the error number.
3. Reference the declared exception name within the corresponding exception-handling routine.

## Functions for Trapping Exceptions

By using two predefined functions, you can decide what subsequent actions to take in an `EXCEPTION` section.

- `SQLERRM` returns character data containing the message associated with the error number.
- `SQLCODE` returns the numeric value for the error code

SQLCODE Value	Description
0	No exception encountered
1	User-defined exception
+100	<code>NO_DATA_FOUND</code> exception
Negative number	Another Oracle server error number



## Functions for Trapping Exceptions

You cannot use `SQLCODE` or `SQLERRM` directly in an SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in the following example:

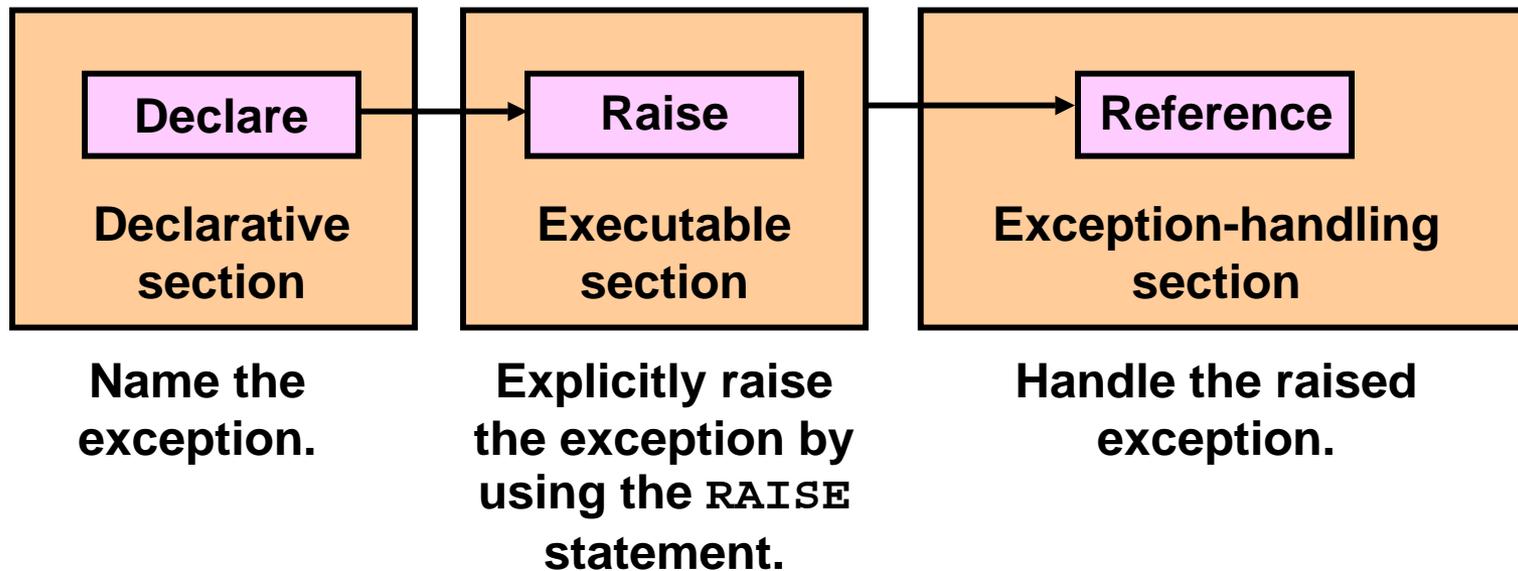
```
DECLARE
    v_error_code      NUMBER;
    v_error_message   VARCHAR2(255);
BEGIN
    ...
EXCEPTION
WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;

    INSERT INTO error_log (e_user,e_date,error_code,error_message)
        VALUES (USER,SYSDATE,v_error_code,v_error_message);
END;
```

# Trapping User-Defined Exceptions

## Trapping User-Defined Exceptions

The third type of exception is user-defined. Within your program you might need to define an error not recognized by the database. Within the PL/SQL block, you name the exception, define the exception, and handle the exception.



## Trapping User-Defined Exceptions (continued)

In the following code, assume that your program prompts the user for a department number and name so that it can update the name of the department.

```
DECLARE
  v_name VARCHAR2(20):='Accounting';
  v_deptno NUMBER := 27;
BEGIN
  UPDATE departments
  SET    department_name = v_name
  WHERE department_id = v_deptno;
END;
```

If department 27 does not exist, the above code does not produce an Oracle error. You need to declare, explicitly raise, and handle a user-defined exception.

## Trapping User-Defined Exceptions (continued)

By including the following code, you define an error that is then handled by the PLSQL compiler. You do this by:

1. Declaring the name of the user-defined exception within the declarative section.
2. Using the `RAISE` statement to raise the exception explicitly within the executable section.
3. Referencing the declared exception within the exception-handling section.



## Trapping User-Defined Exceptions (continued)

The following is the completed code.

```
DECLARE
  e_invalid_department EXCEPTION; ← 1
  v_name VARCHAR2(20):='Accounting';
  v_deptno NUMBER := 27;
BEGIN
  UPDATE departments
    SET    department_name = v_name
    WHERE  department_id = v_deptno;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department; ← 2
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_department ← 3
    THEN DBMS_OUTPUT.PUT_LINE('No such department id. ');
    ROLLBACK;
END;
```



## The RAISE Statement

You can use the RAISE statement to raise a named exception.

You can raise:

- An exception of your own (that is, a user-defined exception)

```
IF v_grand_total=0 THEN
    RAISE e_invalid_total;
ELSE

DBMS_OUTPUT.PUT_LINE(v_num_students/v_grand_total);
END IF;
```

- An Oracle server error

```
IF v_grand_total=0 THEN
    RAISE ZERO_DIVIDE;
ELSE

DBMS_OUTPUT.PUT_LINE(v_num_students/v_grand_total);
END IF;
```



## The RAISE\_APPLICATION\_ERROR Procedure

You can use the `RAISE_APPLICATION_ERROR` procedure to return user-defined error messages from stored subprograms. The main advantage over `RAISE` is that the `RAISE_APPLICATION_ERROR` allows you to associate your own error number and meaningful error message with the exception.

```
RAISE_APPLICATION_ERROR (error_number,  
                          message[, {TRUE | FALSE}]);
```

- *error\_number*: Is a user-specified number for the exception between `-20000` and `-20999` reserved by Oracle for programmer use. These numbers are never used for predefined Oracle server errors.
- *Message*: Is the user-specified message up to 2,048 bytes.
- `TRUE | FALSE`: Is an optional Boolean parameter. (If `TRUE`, the error is placed on the stack of previous errors. If `FALSE`, the default, the error replaces all previous errors.)



## The RAISE\_APPLICATION\_ERROR Procedure (continued)

You can use the RAISE\_APPLICATION\_ERROR in two different places:

- Executable section
- Exception section

```
DECLARE
  v_mgr PLS_INTEGER := 123;
BEGIN
  DELETE FROM employees
    WHERE manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202,
      'This is not a valid manager');
  END IF;
EXCEPTION
  WHEN OTHERS THEN RAISE_APPLICATION_ERROR(-20203,
    'An error has occurred.');
```

```
END;
```

# Recognizing the Scope of Variables and Exceptions



## Nested Blocks

The example shown in the slide has an outer (parent) block (illustrated in blue) and a nested (child) block (illustrated in red). The variable `v_outer_variable` is declared in the outer block and the variable `v_inner_variable` is declared in the inner block.

```
DECLARE
  v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_inner_variable);
    DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```



## Variable Scope

The scope of a variable is the portion of the program in which the variable is declared and is accessible.

The variables `v_father_name` and `v_date_of_birth` are declared in the outer block. Their scope includes the outer and inner blocks.

```
DECLARE
  v_father_name VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
    ...
```

The variables `v_child_name` and `v_date_of_birth` (shown in red) are declared in the inner block and are accessible only within the inner block.



## Variable Visibility

You cannot declare two variables with the same name in the same block. However, you can declare variables with the same name in two different blocks. The two items represented by the identifiers are distinct, and any change in one does not affect the other. It is not recommended to reuse the same name for variables because it can make confusing code. The scope of the `v_date_of_birth` variable declared in the outer block consists of both blocks. However, it is not visible in the inner block because the inner block has a local variable with the same name.

```
DECLARE
  v_father_name VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  ...
```



## Qualifying an Identifier

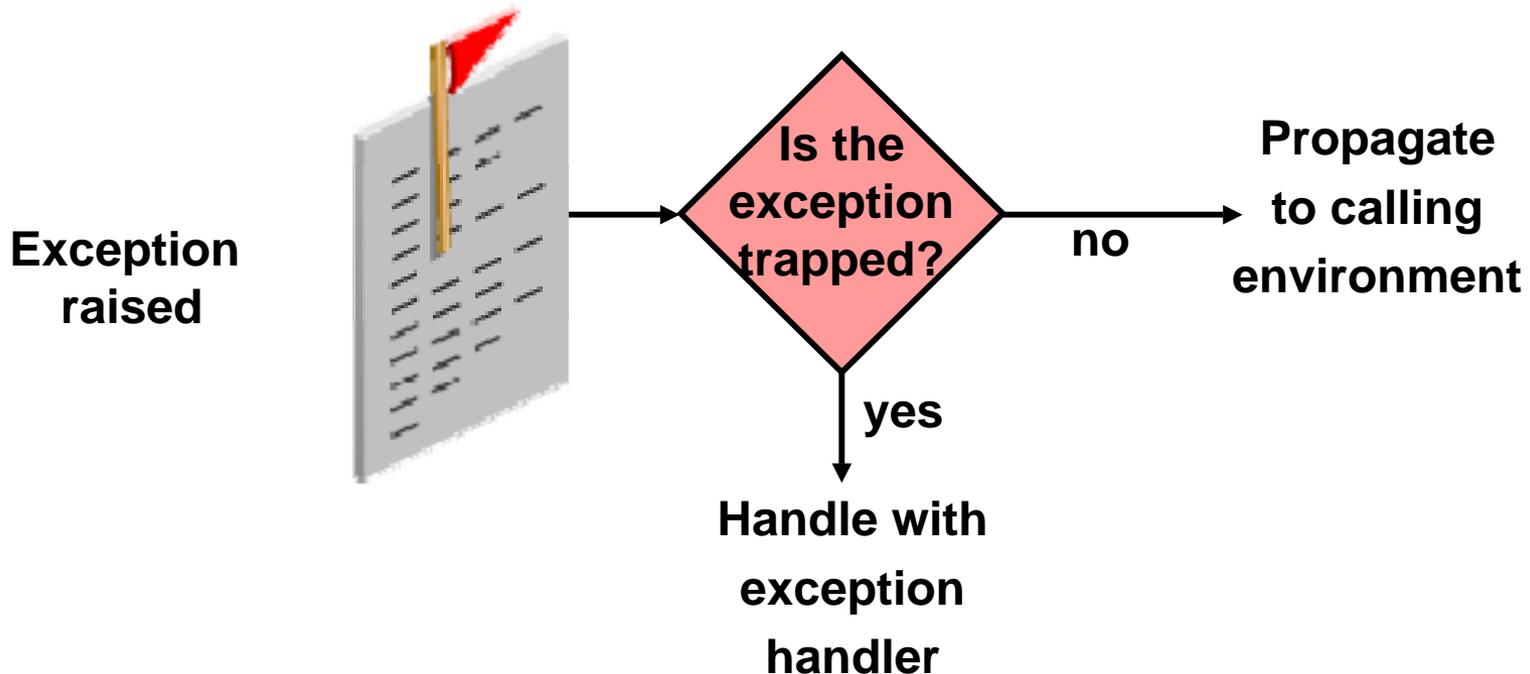
In your code you might want to reference a variable in another block. This can be done by defining a qualifier (label) to a block. You can then use this qualifier to access the variables that have scope but are not visible. In this example, the outer block has the label, <<outer>> .

```
<<outer>>
DECLARE
  v_father_name    VARCHAR2(20):='Patrick';
  v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
    v_child_name    VARCHAR2(20):='Mike';
    v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Date of Birth: `
                          || outer.v_date_of_birth );
  ...
```

## Handling Exceptions

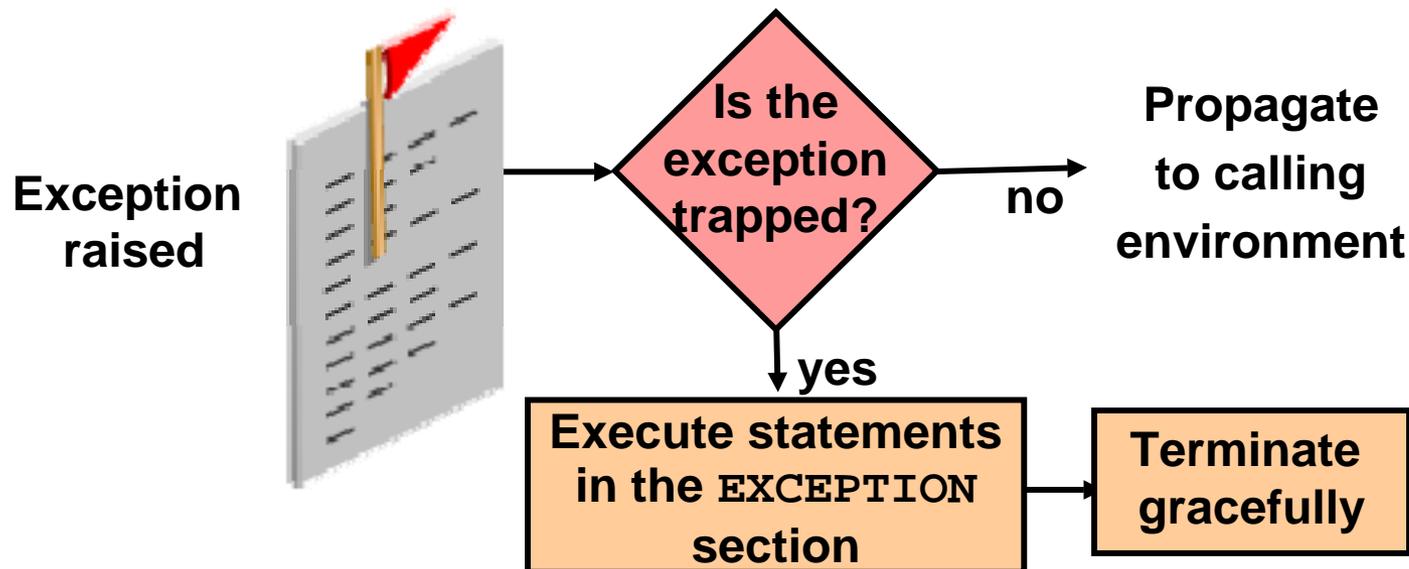
You can handle an exception:

- By trapping it with a handler
- By propagating it to the calling environment



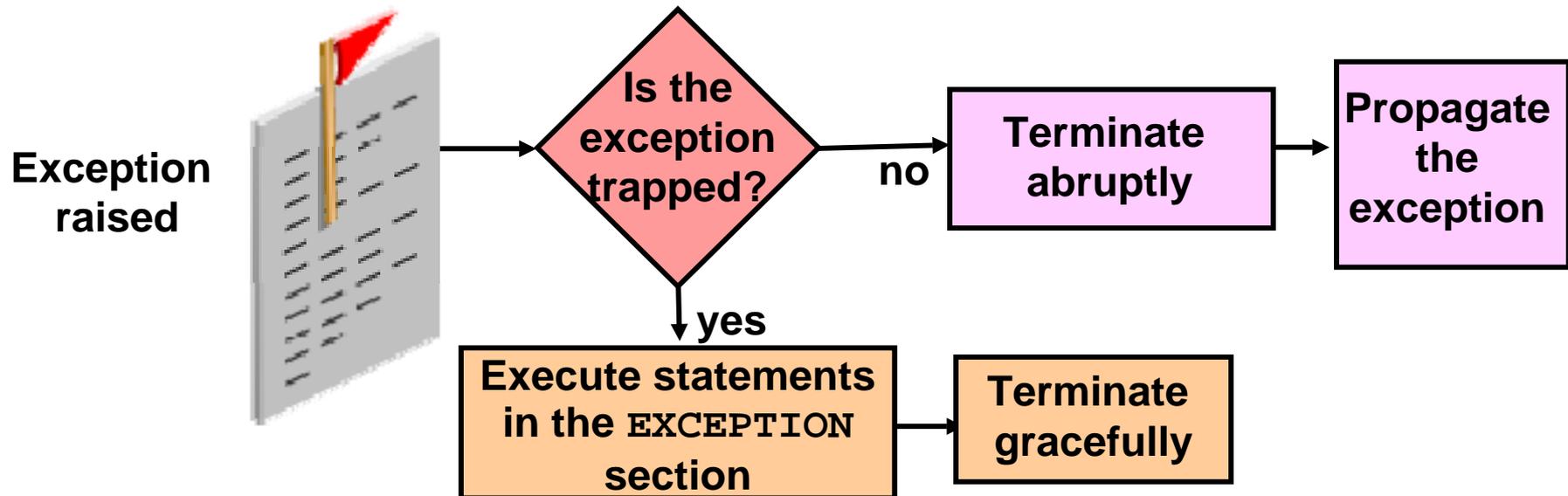
## Trapping Exceptions With a Handler

Include an `EXCEPTION` section in your PL/SQL program to trap exceptions. If the exception is raised in the executable section of the block, processing is handled by the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or to the calling environment. The PL/SQL block terminates successfully.



## Trapping Exceptions With a Handler (continued)

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to an enclosing block or to the calling environment. The calling environment can be any application that is invoking the PL/SQL program.





## Calling Environments

Instead of trapping an exception within the PL/SQL block, propagate the exception to allow the calling environment to handle it. Each calling environment has its own way of displaying and accessing errors.

Application Express	Displays the error number and message to screen
Procedure Builder	Displays the error number and message to screen
Oracle Forms Developer	Accesses error number and message in an <code>ON-ERROR</code> trigger by means of the <code>ERROR_CODE</code> and <code>ERROR_TEXT</code> packaged functions
Precompiler application	Accesses exception number through the <code>SQLCA</code> data structure
An enclosing PL/SQL block	If present, traps exception in exception-handling section of enclosing block

## Propagating Exceptions in a Subblock

If a PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates to successive enclosing blocks until it finds a handler. If none of these blocks handle the exception, an unhandled exception is passed to the host environment.

When the exception propagates to an enclosing block, the remaining executable actions in the inner block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.



## Propagating Exceptions in a Subblock

In the example, the exceptions, `e_no_rows` and `e_integrity`, are declared in the outer block. In the inner block, if the `e_no_rows` exception is raised, PL/SQL looks for the exception to be handled in the subblock. Because the exception is not handled in the subblock, the exception propagates to the outer block, where PL/SQL finds the handler.

```
DECLARE
    . . .
    e_no_rows          exception;
    e_integrity        exception;
    PRAGMA EXCEPTION_INIT (e_integrity,-2292);
BEGIN
    FOR c_record IN emp_cursor LOOP
        BEGIN
            SELECT ...
            UPDATE ...
            IF SQL%NOTFOUND THEN
                RAISE e_no_rows;
            END IF;
        END;
    END LOOP;
EXCEPTION
    WHEN e_integrity THEN ...
    WHEN e_no_rows THEN ...
END;
```

# Creating Procedures

## Differences Between Anonymous Blocks and Subprograms

### Anonymous Blocks

As the name indicates, anonymous blocks are unnamed executable PL/SQL blocks. Because they are unnamed, they cannot be stored in the database for later use. While anonymous blocks can be stored on your PC in a file, the database is not aware of them so no one else can share them.

### Subprograms

Procedures and functions are named PL/SQL blocks, also known as subprograms. These subprograms are compiled and stored in the database. The block structure of the subprograms is much the same as anonymous blocks. While they can be shared, by default they are accessible only by their owner.



## Syntax Differences:

### Anonymous Blocks

```
DECLARE      (Optional)
    Variables, cursors, etc.;
BEGIN        (Mandatory)
    SQL and PL/SQL statements;
EXCEPTION   (Optional)
    WHEN Actions in event of errors;
END;         (Mandatory)
```

### Subprograms (Procedure)

```
CREATE [OR REPLACE] PROCEDURE name [parameters] IS|AS
(Mandatory)
    Variables, cursors, etc.; (Optional)
BEGIN      (Mandatory)
    SQL and PL/SQL statements;
EXCEPTION (Optional)
    WHEN Actions in event of errors;
END [name]; (Mandatory)
```



## Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled on every execution	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	They are named and therefore can be invoked by other applications
Do not return values	Subprograms called functions must return values
Cannot take parameters	Can take parameters



## Benefits of Subprograms

Procedures and functions have many benefits due to the modularizing of the code:

- Easy maintenance — Modifications need only be done once to improve multiple applications and minimize testing.
- Code Reuse — Subprograms are located in one place. When compiled and validated, they can be used and reused in any number of applications.
- Improved data security -- Indirect access to database objects from unauthorized users can be permitted with security privileges. By default, subprograms run with the privileges of the subprogram owner, not the privileges of the user.

## Benefits of Subprograms (continued)

- Data integrity — Related actions can be grouped into a block and are performed together (“Statement Processed”) or not at all.
- Improved performance — Compiled PL/SQL code that is stored in the shared SQL area cache of the server can be reused. Subsequent calls to the subprogram avoid compiling the code again. Also, many users can share a single copy of the code in memory.
- Improved code clarity — By using appropriate names and conventions to describe the action of the routines, you can reduce the need for comments, and enhance the clarity of the code.

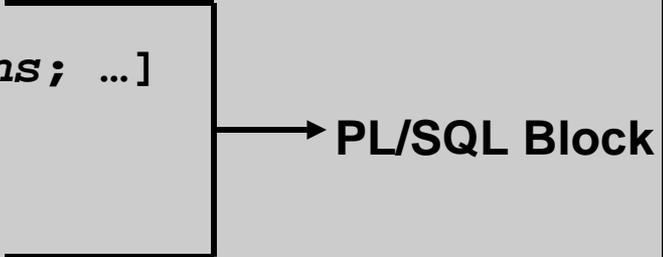
## What Is a Procedure?

- A procedure is a named PL/SQL block that can accept parameters.
- Generally, you use a procedure to perform an action (sometimes called a “side-effect”).
- A procedure is compiled and stored in the database as a schema object.
  - Shows up in `USER_OBJECTS` as an object type of `PROCEDURE`
  - More details in `USER_PROCEDURES`
  - Detailed PL/SQL code in `USER_SOURCE`



# Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter1 [mode1] datatype1,
    parameter2 [mode2] datatype2,
    . . .)]
IS|AS
  [local_variable_declarations; ...]
BEGIN
  -- actions;
END [procedure_name];
```



- Parameters are optional
- Mode defaults to `IN`
- Datatype can be either explicit (for example, `VARCHAR2`) or implicit with `%TYPE`
- Body is the same as an anonymous block

 **Syntax (continued)**

- Use `CREATE PROCEDURE` followed by the name, optional parameters, and keyword `IS` or `AS`.
- Add the `OR REPLACE` option to overwrite an existing procedure.
- Write a PL/SQL block containing local variables, a `BEGIN`, and an `END` (or `END procedure_name`).

 **Example**

In the following example, the procedure inserts a new department into the `dept` table. The procedure declares two variables, `v_dept_id` and `v_dept_name`. The procedure uses the `SQL%ROWCOUNT` SQL attribute to check whether the row was successfully inserted.

```
CREATE OR REPLACE PROCEDURE add_dept
IS
    v_dept_id    dept.department_id%TYPE;
    v_dept_name  dept.department_name%TYPE;
BEGIN
    v_dept_id    := 280;
    v_dept_name  := 'Training';
    INSERT INTO dept(department_id,department_name)
        VALUES(v_dept_id,v_dept_name);
    DBMS_OUTPUT.PUT_LINE('Inserted ' || SQL%ROWCOUNT || 'row');
END;
```

## Invoking Procedures

You can invoke (execute) a procedure from:

- An anonymous block
- Another procedure
- A calling application

Note: You cannot invoke a procedure from inside an SQL statement, such as `SELECT`.

### Invoking a Procedure from Application Express

To invoke a procedure in Oracle Application Express, write and run a small anonymous block that invokes the procedure.

For example:

```
BEGIN
    add_dept ;
END;
```

## Correcting Errors in Create Procedure Statements

If compilation errors exist, Application Express display them in the output portion of the SQL Commands window. You then edit the source code to make corrections.

After you have corrected the code, you must recreate the procedure. There are two ways to do this:

- Use a `CREATE OR REPLACE PROCEDURE` statement to overwrite the existing code (most common).
- `DROP` the procedure first and then execute the `CREATE PROCEDURE` statement (less common).



## Saving Your Work

When a procedure has been created successfully, you should save its definition in case you need to modify the code later.

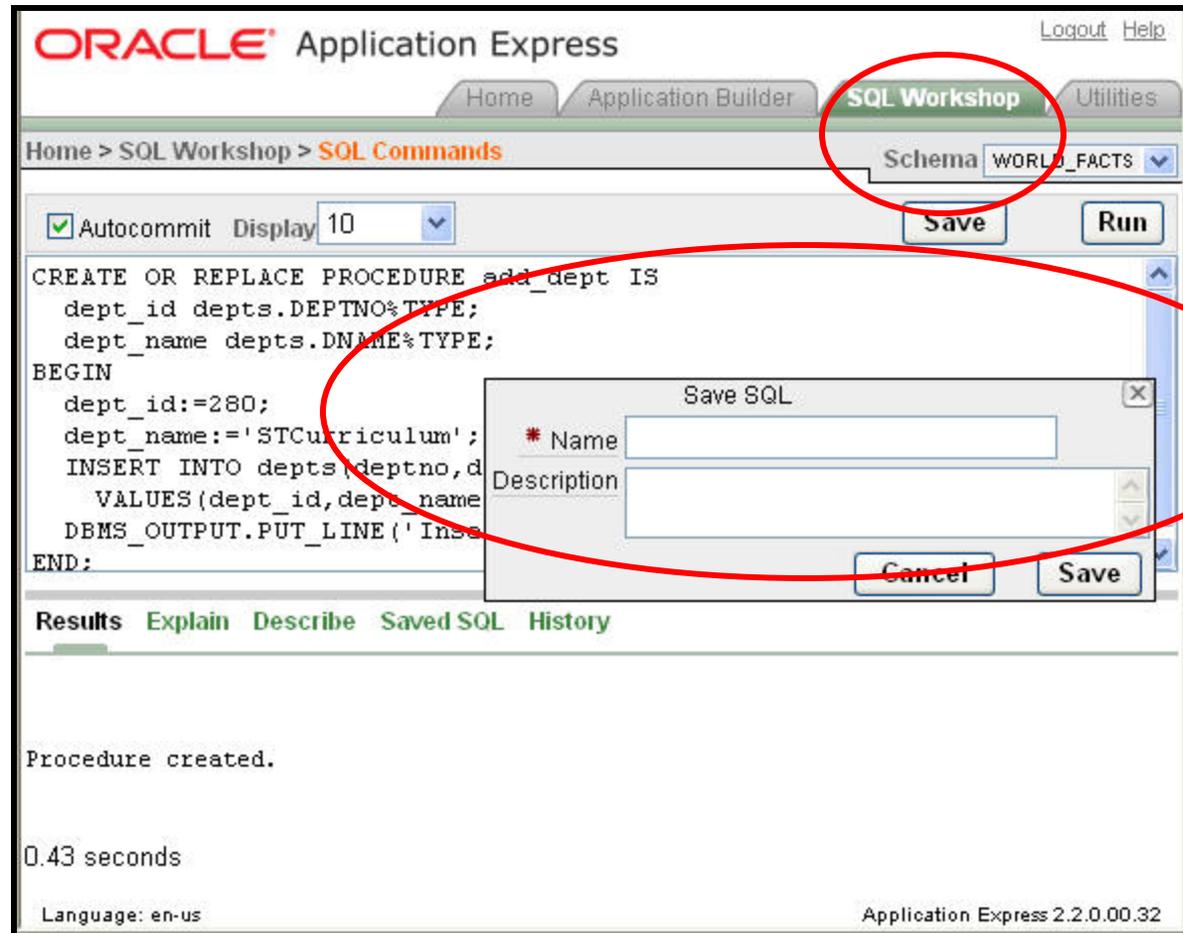
The screenshot displays the Oracle Application Express interface. The top navigation bar includes 'Home', 'Application Builder', 'SQL Workshop', and 'Utilities'. The current page is 'SQL Commands' for the 'WORLD\_FACTS' schema. The 'Autocommit' checkbox is checked, and the 'Display' setting is set to 10. The 'Save' and 'Run' buttons are visible. The SQL editor contains the following code:

```
CREATE OR REPLACE PROCEDURE add_dept IS
  dept_id depts.DEPTNO%TYPE;
  dept_name depts.DNAME%TYPE;
BEGIN
  dept_id:=280;
  dept_name:='STCurriculum';
  INSERT INTO depts(deptno,dname)
    VALUES(dept_id,dept_name);
  DBMS_OUTPUT.PUT_LINE('Inserted '||SQL%ROWCOUNT||'row');
END;
```

Below the editor, the 'Results' tab is selected, showing the output: 'Procedure created.' and '0.43 seconds'. The 'Language' is set to 'en-us' and the version is 'Application Express 2.2.0.00.32'. A red oval highlights the 'Results' section.

## Saving Your Work (continued)

In the Application Express SQL Commands window, click the Save button and enter a name and optional description for your code.



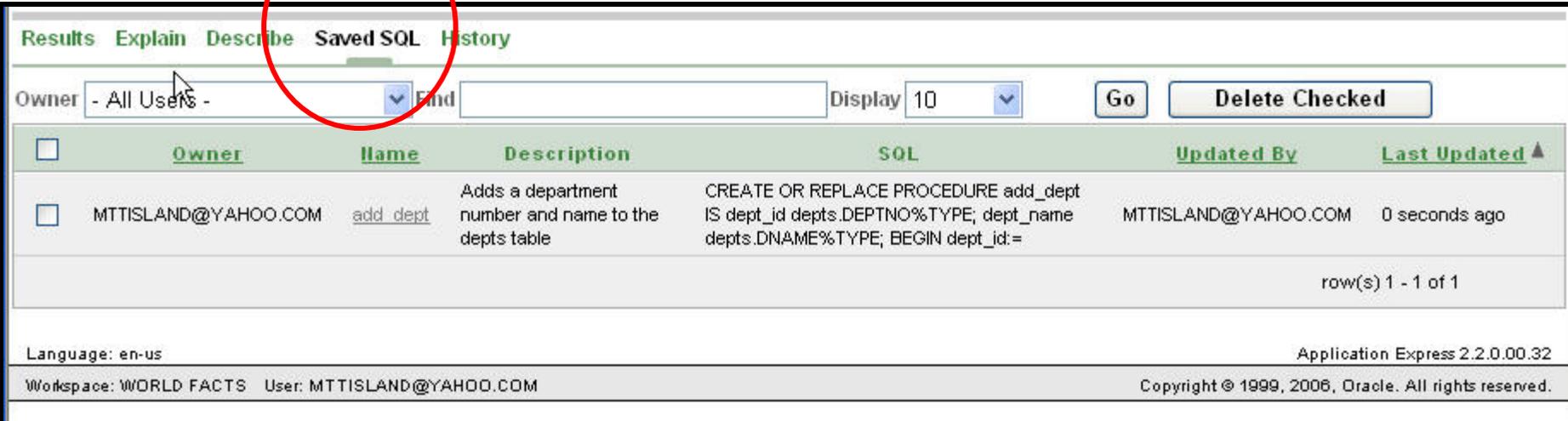
The screenshot shows the Oracle Application Express SQL Workshop interface. The 'SQL Workshop' tab is selected and circled in red. The 'Save' button is also circled in red. A 'Save SQL' dialog box is open, prompting for a name and description. The SQL code in the background is as follows:

```
CREATE OR REPLACE PROCEDURE add_dept IS
  dept_id depts.DEPTNO%TYPE;
  dept_name depts.DNAME%TYPE;
BEGIN
  dept_id:=280;
  dept_name:='STCurriculum';
  INSERT INTO depts (deptno, dname)
  VALUES (dept_id, dept_name)
  DBMS_OUTPUT.PUT_LINE ('Inserted');
END;
```

The results pane shows "Procedure created." and "0.43 seconds". The footer indicates "Language: en-us" and "Application Express 2.2.0.00.32".

## Saving Your Work (continued)

You can view and reload your code later by clicking the Saved SQL button in the SQL Commands window.



Results Explain Describe **Saved SQL** History

Owner: - All Users - Find: Display: 10 Go Delete Checked

<input type="checkbox"/>	Owner	Name	Description	SQL	Updated By	Last Updated
<input type="checkbox"/>	MTTISLAND@YAHOO.COM	add_dept	Adds a department number and name to the depts table	CREATE OR REPLACE PROCEDURE add_dept IS dept_id depts.DEPTNO%TYPE; dept_name depts.DNAME%TYPE; BEGIN dept_id:=	MTTISLAND@YAHOO.COM	0 seconds ago

row(s) 1 - 1 of 1

Language: en-us Application Express 2.2.0.00.32

Workspace: WORLD FACTS User: MTTISLAND@YAHOO.COM Copyright © 1999, 2006, Oracle. All rights reserved.

## Alternative Tools for Developing Procedures

If you end up writing PL/SQL procedures for a living, there are other free tools that make this process easier than using Notepad as the editor. For example, Oracle tools, such as SQL Developer and JDeveloper assist you by:

- Color-coding **commands** vs **variables** vs **constants**
- Highlighting matched and mismatched **((parentheses)**
- Displaying errors more graphically
- “Prettifying” code with standard indentations and capitalization
- Completing commands when typing
- Completing column names from tables

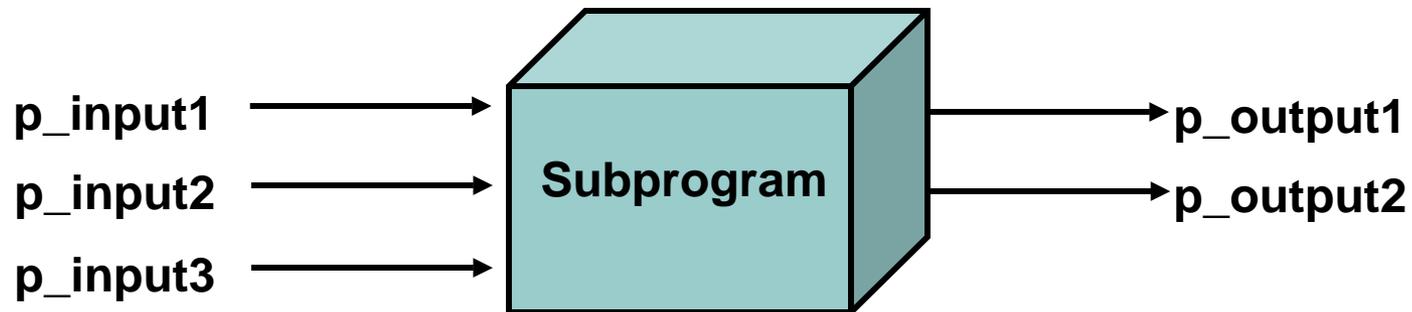
# Using Parameters in Procedures

## What Are Parameters?

Parameters pass or communicate data between the caller and the subprogram.

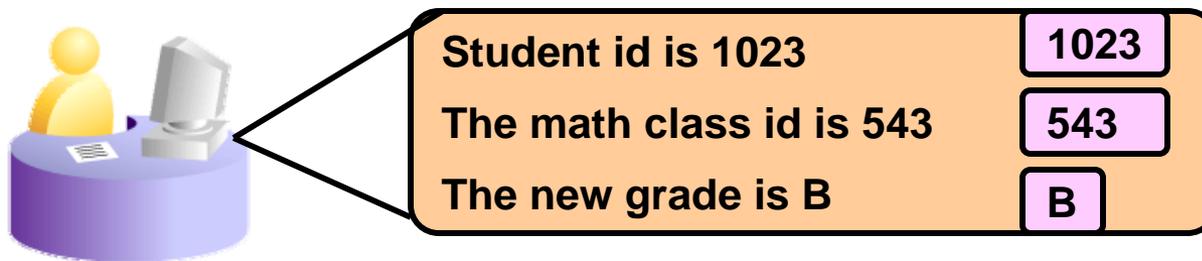
You can think of parameters as a special form of variables, whose input values are initialized by the calling environment when the subprogram is called, and whose output values are returned to the calling environment when the subprogram returns control to the caller.

By convention, parameters are often named with a “p\_” prefix.



## What Are Parameters? (continued)

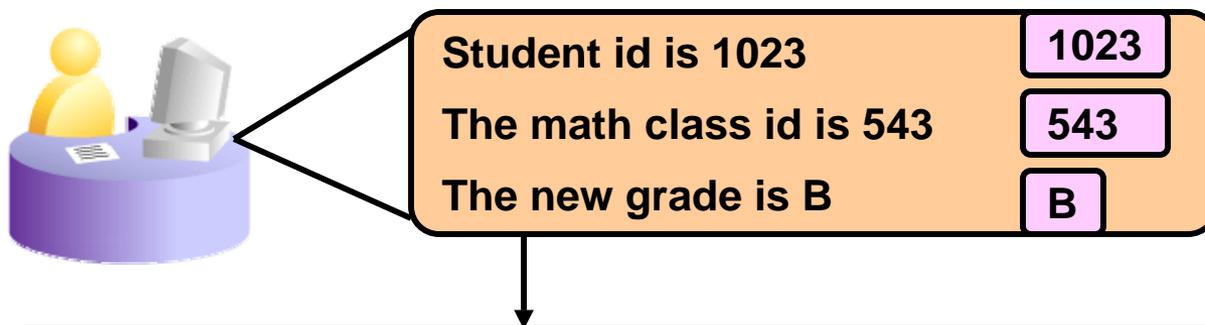
Consider the following example where a math teacher needs to change a student's grade from a C to a B in the student administration system.



The calling system is passing values for student id, class id, and grade to a subprogram.

## What Are Parameters? (continued)

The `change_grade` procedure accepts three parameters: `p_student_id`, `p_class_id`, and `p_grade`. These parameters act like local variables in the `change_grade` procedure.



```
PROCEDURE change_grade (p_student_id IN VARCHAR2,  
p_class_id IN VARCHAR2, p_grade IN VARCHAR2) IS  
BEGIN  
...  
    UPDATE grade_table SET grade= B  
        WHERE student_id= 1023 AND class_id= 543 ;  
...  
END;
```

## What Are Arguments?

Parameters are commonly referred to as arguments. However, arguments are more appropriately thought of as the actual values assigned to the parameter variables when the subroutine is called at runtime.

In the previous example, 1023 is an argument passed in to the `p_student_id` parameter.

Student id is 1023	1023
The math class id is 543	543
The new grade is B	B

Even though parameters are a kind of variable, the IN parameter arguments act as constants and cannot be changed by the subprogram.

## Creating Procedures With Parameters

The example shows a procedure with two parameters. Running this first statement creates the `raise_salary` procedure in the database. The second example executes the procedure, passing the arguments 176 and 10 to the two parameters.

```
CREATE OR REPLACE PROCEDURE raise_salary
(p_id      IN employees.employee_id%TYPE,
 p_percent IN NUMBER)
IS
BEGIN
    UPDATE employees
    SET     salary = salary * (1 + p_percent/100)
    WHERE  employee_id = p_id;
END raise_salary;
```

```
BEGIN raise_salary(176,10); END;
```

## Invoking Procedures With Parameters

To invoke a procedure from Application Express, create an anonymous block and use a direct call inside the executable section of the block. Where you want to call the new procedure, enter the procedure name and parameter values (arguments). For example:

```
BEGIN
  raise_salary (176, 10);
END;
```

The arguments must be entered in the same order as they are declared in the procedure.

## Invoking Procedures With Parameters (continued)

To invoke a procedure from another procedure, use a direct call inside an executable section of the block. At the location of calling the new procedure, enter the procedure name and parameter's arguments.

```
CREATE OR REPLACE PROCEDURE process_employees
IS
    CURSOR emp_cursor IS
        SELECT employee_id
        FROM employees;
BEGIN
    FOR v_emp_rec IN emp_cursor
    LOOP
        raise_salary(v_emp_rec.employee_id, 10);
    END LOOP;
    COMMIT;
END process_employees;
```

## Types of Parameters

There are two types of parameters: **Formal** and **Actual**.

A parameter-name declared in the procedure heading is called a formal parameter. The corresponding parameter-name (or value) in the calling environment is called an actual parameter.

```
CREATE OR REPLACE PROCEDURE fetch_emp
  (p_emp_id IN employees.employee_id%TYPE) IS ...
END;
...
fetch_emp(v_emp_id);
```

`p_emp_id` is a formal parameter and `v_emp_id` is an actual parameter.



## Formal Parameters

Formal parameters are variables that are declared in the parameter list of a subprogram specification. In the following example, in the procedure `raise_sal`, the identifiers `p_id` and `p_sal` represent formal parameters.

```
CREATE PROCEDURE raise_sal
    (p_id IN NUMBER, p_sal IN NUMBER) IS
BEGIN ...
END raise_sal;
```

Notice that the formal parameter datatypes do not have sizes. For instance `p_sal` is `NUMBER`, not `NUMBER(6,2)`.



## Actual Parameters

Actual parameters can be literal values, variables, or expressions that are provided in the parameter list of a called subprogram. In the following example, a call is made to `raise_sal`, where the `a_emp_id` variable provides the actual parameter value for the `p_id` formal parameter and 2000 is supplied as the actual parameter value for `p_sal`.

```
a_emp_id := 100;  
raise_sal(a_emp_id, 2000);
```

Actual parameters:

- Are associated with formal parameters during the subprogram call
- Can also be expressions, as in the following example:  
`raise_sal(a_emp_id, raise+100);`



## Formal and Actual Parameters

The formal and actual parameters should be of compatible data types. If necessary, before assigning the value, PL/SQL converts the data type of the actual parameter value to that of the formal parameter.

For example, you can pass in a salary of '1000.00' in single quotes, so it is coming in as the *letter* 1 and the *letters* zero etc., which gets converted into the *number* one thousand. This is slower and should be avoided if possible.

You can find out the datatypes that are expected by using the command `DESCRIBE proc_name`.

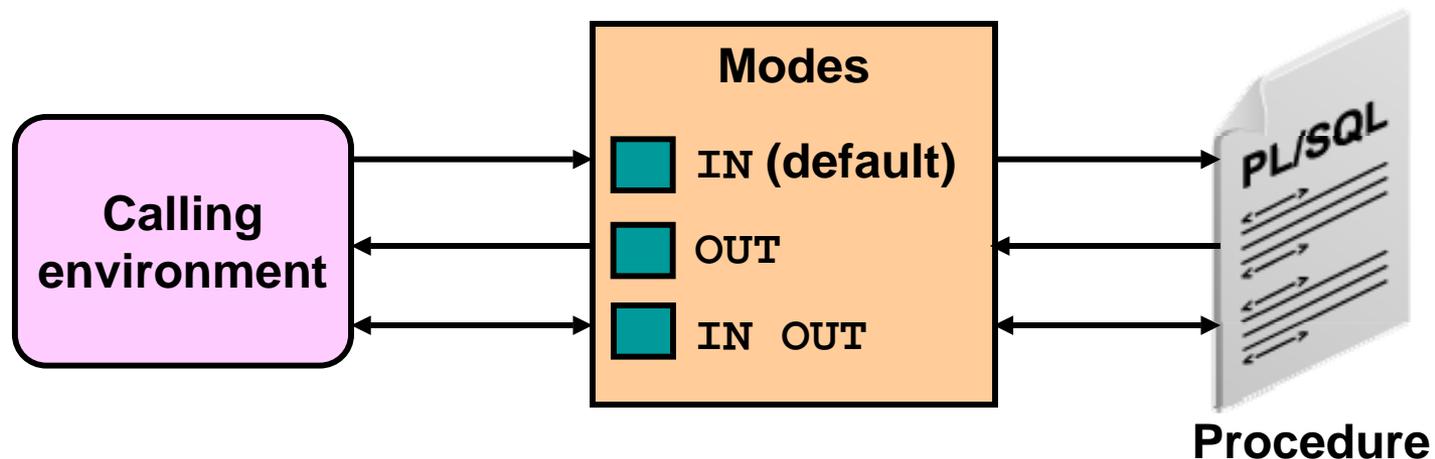
# Passing Parameters

## Procedural Parameter Modes

Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.

Parameter-passing modes:

- An `IN` parameter (the default) provides values for a subprogram to process.
- An `OUT` parameter returns a value to the caller.
- An `IN OUT` parameter supplies an input value, which can be returned (output) as a modified value.





## The IN mode Is the Default if No Mode Is Specified

```
CREATE PROCEDURE procedure(param [mode] datatype)  
...
```

```
CREATE OR REPLACE PROCEDURE raise_salary  
  (p_id      IN employees.employee_id%TYPE,  
   p_percent IN NUMBER)  
IS  
BEGIN  
  UPDATE employees  
    SET    salary = salary * (1 + p_percent/100)  
    WHERE employee_id = p_id;  
END raise_salary;
```



## Using OUT Parameters: Example

```
CREATE OR REPLACE PROCEDURE query_emp
(p_id      IN  employees.employee_id%TYPE,
p_name    OUT employees.last_name%TYPE,
p_salary  OUT employees.salary%TYPE) IS
BEGIN
  SELECT  last_name, salary INTO p_name, p_salary
  FROM    employees
  WHERE   employee_id = p_id;
END query_emp;
```

```
DECLARE
  a_emp_name employees.last_name%TYPE;
  a_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(178, a_emp_name, a_emp_sal); ...
END;
```

The procedure accepts the value 178 for employee ID and retrieves the name and salary of the employee with ID 178 into the two OUT parameters.



## Viewing OUT Parameters in Application Express

Use PL/SQL variables that are displayed with calls to the `DBMS_OUTPUT.PUT_LINE` procedure.

```
DECLARE
  a_emp_name employees.last_name%TYPE;
  a_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(178, a_emp_name, a_emp_sal);
  DBMS_OUTPUT.PUT_LINE('Name: ' || a_emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || a_emp_sal);
END;
```

Name: Grant

Salary: 7700

Attempting to use or read `OUT` parameters inside the procedure that declares them results in a compilation error. The `OUT` parameters can be assigned values only in the body of the procedure in which they are declared.

 **Using IN OUT Parameters:**

phone\_no <before the call>

'8006330575'

phone\_no <after the call>

'(800)633-0575'

```
CREATE OR REPLACE PROCEDURE format_phone
  (p_phone_no IN OUT VARCHAR2) IS
BEGIN
  p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
                ')' || SUBSTR(p_phone_no,4,3) ||
                '-' || SUBSTR(p_phone_no,7);
END format_phone;
```

Using an IN OUT parameter, you pass a value into a procedure that can be updated within the procedure. The actual parameter value supplied from the calling environment can return as either the original unchanged value or a new value that is set within the procedure. An IN OUT parameter acts as an initialized variable.



## Using the previous IN OUT example

The following code creates an anonymous block that declares `a_phone_no`, assigns the unformatted phone number to it, and passes it as an actual parameter to the `FORMAT_PHONE` procedure. The procedure is executed and returns an updated string in the `a_phone_no` variable, which is then displayed.

```
DECLARE
    a_phone_no VARCHAR2(13);
BEGIN
    a_phone_no := '8006330575' ;
    format_phone (a_phone_no);
    DBMS_OUTPUT.PUT_LINE('The formatted phone
        number is: ` || a_phone_no);
END;
```



## Summary of Parameter Modes

<b>IN</b>	<b>OUT</b>	<b>IN OUT</b>
<b>Default mode</b>	<b>Must be specified</b>	<b>Must be specified</b>
<b>Value is passed into subprogram</b>	<b>Returned to calling environment</b>	<b>Passed into subprogram; returned to calling environment</b>
<b>Formal parameter acts as a constant</b>	<b>Uninitialized variable</b>	<b>Initialized variable</b>
<b>Actual parameter can be a literal, expression, constant, or initialized variable</b>	<b>Must be a variable</b>	<b>Must be a variable</b>
<b>Can be assigned a default value</b>	<b>Cannot be assigned a default value</b>	<b>Cannot be assigned a default value</b>

## Syntax for Passing Parameters

There are three ways of passing parameters from the calling environment:

- **Positional:** Lists the actual parameters in the same order as the formal parameters
- **Named:** Lists the actual parameters in arbitrary order and uses the association operator ( '`=>`' which is an equal and an arrow together) to associate a named formal parameter with its actual parameter
- **Combination:** Lists some of the actual parameters as positional (no special operator) and some as named (with the `=>` operator). When using the combination notation, positional notation parameters must be listed before named notation parameters.

You must provide a value for each parameter unless the formal parameter is assigned a default value.



## Parameter Passing: Examples

```
CREATE OR REPLACE PROCEDURE add_dept(  
  p_name IN departments.department_name%TYPE,  
  p_loc  IN departments.location_id%TYPE) IS  
BEGIN  
  INSERT INTO departments(department_id,  
                          department_name, location_id)  
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);  
END add_dept;
```

- Passing by positional notation:

```
add_dept ('EDUCATION', 1400);
```

- Passing by named notation:

```
add_dept (p_loc=>1400, p_name=>'EDUCATION');
```

- Passing by combination notation:

```
add_dept ('EDUCATION', p_loc=>1400);
```



## Using the DEFAULT Option for IN Parameters

You can assign a default value for formal IN parameters. This provides flexibility when passing parameters.

```
CREATE OR REPLACE PROCEDURE add_dept(  
  p_name departments.department_name%TYPE := 'Unknown',  
  p_loc  departments.location_id%TYPE  DEFAULT 1400)  
IS  
BEGIN  
  INSERT INTO departments (...)  
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);  
END add_dept;
```

The code shows two ways of assigning a default value to an IN parameter. The two ways are the assignment operator (`:=`) or the `DEFAULT` option.

## Using the DEFAULT Option for Parameters

The following are three ways of invoking the `add_dept` procedure:

- The first example assigns the default values for each parameter.
- The second example illustrates a combination of position and named notation to assign values. In this case, using named notation is presented as an example.
- The last example uses the default value for the `p_name` parameter and the supplied value for the `p_loc` parameter.

```
add_dept;  
add_dept ('ADVERTISING', p_loc => 1400);  
add_dept (p_loc => 1200);
```



## Guidelines for Using the **DEFAULT** Option for Parameters

- You cannot assign default values to **OUT** and **IN OUT** parameters *in the header*, but you can in the body of the procedure.
- Usually, you can use named notation to override the default values of formal parameters. However, you cannot skip providing an actual parameter if there is no default value provided for a formal parameter.
- A parameter inheriting a **DEFAULT** value is different from **NULL**.

# Creating Functions



## What Is a Stored Function?

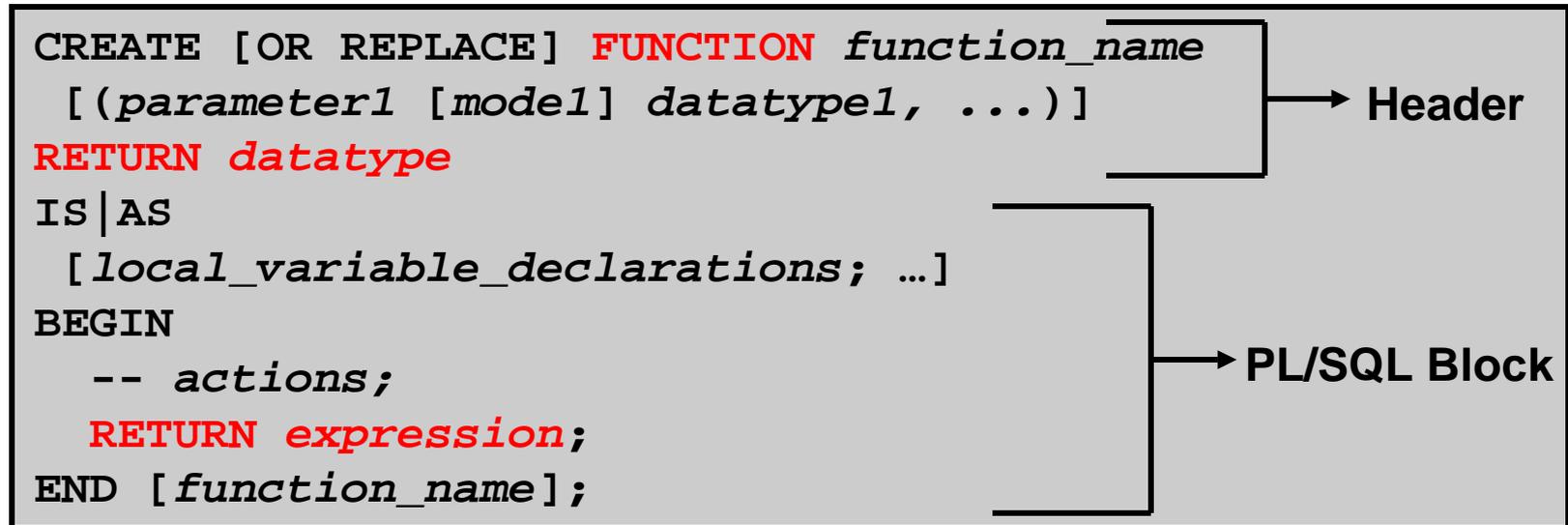
- A function is a named PL/SQL block (a subprogram) that can accept optional `IN` parameters and must return a single value.
- In general, you use a function to compute a value.
- Functions are stored in the database as schema objects for repeated execution.

## What Is a Stored Function? (continued)

- A function can be called as part of an SQL expression or as part of a PL/SQL expression.
  - Certain return types, for example, Boolean, prevent a function from being called as part of a `SELECT`.
- In SQL expressions, a function must obey specific rules to control side effects. Side effects to be avoided are:
  - Any kind of DML or DDL
  - `COMMIT` or `ROLLBACK`
  - Altering global variables
- In PL/SQL expressions, 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.



The header is like a PROCEDURE header with two differences:

1. The mode should only be IN
2. The RETURN clause is instead of an OUT mode.

## Syntax for Creating Functions (continued)

- A function is a PL/SQL block that returns a single value. A RETURN statement must be provided to return a value with a data type that is consistent with the function declaration type.
- You create new functions with the CREATE [OR REPLACE] FUNCTION statement, which can declare a list of parameters, must return exactly one value, and must define the actions to be performed by the standard PL/SQL block.

## Stored Function **With a Parameter**

- Create the function:

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

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

```
... v_salary := get_sal(100);
```



## Returns Can Be in Both BEGIN and/or From EXCEPTION Sections

```
CREATE OR REPLACE FUNCTION get_sal
(p_id employees.employee_id%TYPE)
RETURN NUMBER IS
v_sal employees.salary%TYPE := 0;
BEGIN
SELECT salary INTO v_sal
FROM employees WHERE employee_id = p_id;
RETURN v_sal;
EXCEPTION
WHEN NO_DATA_FOUND THEN RETURN NULL;
END get_sal;
```

## Ways to Invoke (or Execute) Functions With Parameters

- Invoke as part of a PL/SQL expression, using a local variable to store the returned result:

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

A

- Use as a parameter to another subprogram:

```
... DBMS_OUTPUT.PUT_LINE(get_sal(100));
```

B

- Use in an SQL statement (subject to restrictions):

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

C

## Ways to Call Functions With Parameters

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

- As part of PL/SQL expressions: (A) uses a local variable in an anonymous block to hold the returned value from a function.
- As a parameter to another subprogram: (B) demonstrates this usage. The `get_sal` function with all its arguments is nested in the parameter required by the `DBMS_OUTPUT.PUT_LINE` procedure.
- As an expression in a SQL statement: (C) shows how a function can be used as a single-row function in an SQL statement.

## Invoking Functions Without Parameters

Most functions have parameters, but they are optional. The following are system functions USER and SYSDATE without parameters.

- Invoke as part of a PL/SQL expression, using a local variable to obtain the result:

```
DECLARE v_today DATE;  
BEGIN  
    v_today := SYSDATE; ...  
END;
```

- Use as a parameter to another subprogram:

```
... DBMS_OUTPUT.PUT_LINE(USER);
```

- Use in an SQL statement (subject to restrictions):

```
SELECT job_id, SYSDATE-hiredate FROM employees;
```

## Benefits and Restrictions That Apply to Functions

Try things quickly: Functions allow you to temporarily display a value in a new format: a different case, annually vs monthly (times 12), concatenated, or with substrings.

Extend functionality: Add new features, such as spell checking and parsing.

Restrictions: PL/SQL types do not completely overlap with SQL types. What is fine for PL/SQL (for example, `BOOLEAN`, `RECORD`) is invalid for a `SELECT`.

Restrictions: PL/SQL sizes are not the same as SQL sizes. For example, a PL/SQL `VARCHAR2` variable can be up to 32 KB, whereas an SQL `VARCHAR2` column can be only up to 4 KB.



# Differences Between Procedures and Functions

## Procedures

```
CREATE [OR REPLACE] PROCEDURE name [parameters] IS|AS (Mandatory)
    Variables, cursors, etc. (Optional)
BEGIN (Mandatory)
    SQL and PL/SQL statements;
EXCEPTION (Optional)
    WHEN exception-handling actions;
END [name]; (Mandatory)
```

## Functions

```
CREATE [OR REPLACE] FUNCTION name [parameters] (Mandatory)
    RETURN datatype IS|AS (Mandatory)
    Variables, cursors, etc. (Optional)
BEGIN (Mandatory)
    SQL and PL/SQL statements;
    RETURN ...; (One Mandatory, more optional)
EXCEPTION (Optional)
    WHEN exception-handling actions;
END [name]; (Mandatory)
```



## Differences/Similarities Between Procedures and 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

Both can have zero or more `IN` parameters that can be transferred to and from the calling environment.

Both have the standard block structure including exception handling.

## Differences Between Procedures and Functions

### Procedures

- You create a procedure to store a series of actions for later execution. 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 might be better rewritten as a function returning the value.

### Functions

- You create a function when you want to compute a value that must be returned to 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` modes. Although a function using `OUT` can be used in a PL/SQL procedure or block, it cannot be used in SQL statements.

# Using Functions in SQL Statements

## Advantages of Functions in SQL Statements

- In the `WHERE` clause of a `SELECT` statement, functions can increase efficiency by eliminating unwanted rows before the data is sent to the application
- Can manipulate data values
- User-defined functions in particular can extend SQL where activities are too complex, too awkward, or unavailable with regular SQL



## Functions in SQL Expressions:

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (p_value * 0.08);
END tax;
```

Function created:

```
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 50;
```

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
124	Mourgos	5800	464
141	Rajs	3500	280
142	Davies	3100	248
143	Matos	2600	208
144	Vargas	2500	200

## 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 `ORDER BY` and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement.

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

```
SELECT employee_id, tax(salary)
FROM employees
WHERE tax(salary) > (SELECT MAX(tax(salary))
                     FROM employees
                     WHERE department_id = 30)
ORDER BY tax(salary) DESC;
```



## Restrictions on Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
  - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types, such as `BOOLEAN` or `%ROWTYPE`
  - Not exceed SQL size limits (`VARCHAR2` of 32 KB versus 4 KB)
  - Return valid SQL data types, not PL/SQL-specific types (same as above)
- When calling functions in SQL statements:
  - Parameters must be specified with positional notation. Named notation (`=>`) is not allowed.

## 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 cannot query or contain DML on the same table
- SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)
- SQL statements cannot issue DDL (for example, `CREATE TABLE`) or DCL (for example, `ALTER SESSION`) because they also do an implicit `COMMIT`

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



## Restrictions on Calling Functions From SQL:

```
CREATE OR REPLACE FUNCTION dml_call_sql(p_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', p_sal);
  RETURN (p_sal + 100);
END;
```

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

ORA-04091: table USVA\_TEST\_SQL01\_S01.EMPLOYEES is mutating, trigger/function may not see it



## Restrictions on Calling Functions From SQL: Example

- In the following example, the `query_call_sql` function queries the `EMPLOYEES` table:

```
CREATE OR REPLACE FUNCTION query_max_sal (p_dept_id NUMBER)
RETURN NUMBER IS
  v_num NUMBER;
BEGIN
  SELECT MAX(salary) INTO v_num FROM employees
    WHERE department_id = p_dept_id;
  RETURN (v_num);
END;
```

- When invoked from the following DML statement, it returns the “mutating table” error message similar to the error message shown in the previous slide:

```
UPDATE employees SET salary = query_max_sal(department_id)
  WHERE employee_id = 170;
```

# Review of the Data Dictionary

## How Can You Read the Dictionary?

There are three classes of table from which you can `SELECT` to view information from the Dictionary:

- The `USER_*` tables contain information about objects that you own, usually because you created them. Examples: `USER_TABLES`, `USER_INDEXES`.
- The `ALL_*` tables contain information about objects that you have privileges to use. These include the `USER_*` information as a subset, because you always have privileges to use the objects that you own. Examples: `ALL_TABLES`, `ALL_INDEXES`.
- The `DBA_*` tables contain information about everything in the database, no matter who owns them. Normally, only the Database Administrator can use the `DBA_*` tables. Examples: `DBA_TABLES`, `DBA_INDEXES`.



## Viewing Information in the Dictionary

Although you are not allowed to modify the dictionary yourself, you can `DESCRIBE` and `SELECT` from Dictionary tables:

```
DESCRIBE ALL_TABLES
```

```
SELECT table_name, owner FROM ALL_TABLES;
```

```
SELECT object_type, object_name FROM USER_OBJECTS;
```

```
SELECT object_type, COUNT(*) FROM USER_OBJECTS  
GROUP BY object_type;
```

## Using the Super-View Dictionary

There are several hundred Dictionary tables and no-one can remember the names of all of them. You don't have to!

There is a super-view called `DICTIONARY` (or `DICT` for short), which lists all the Dictionary tables.

You can use `DICT` like a Web Search engine (such as Google) to show the names and descriptions (comments) of a relevant subset of Dictionary tables.

The next slide shows how to do this.



## Using the Super-View Dictionary

First try:

```
SELECT COUNT(*) FROM DICT WHERE table_name LIKE 'USER%';
```

You will see that there are more than a hundred `USER_*` tables.

Can you remember which one of them shows you information about which table columns are indexed? Most people can't!

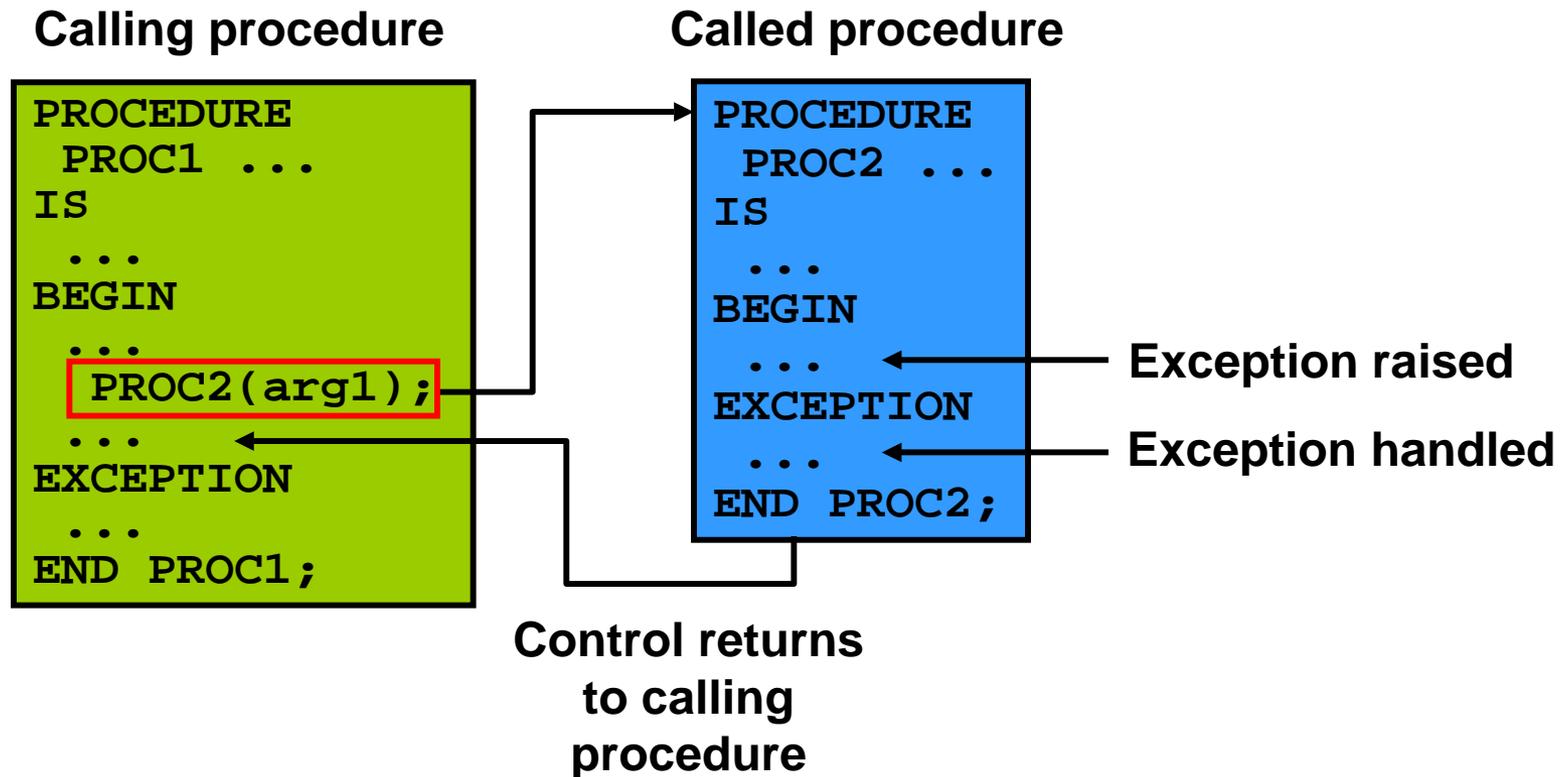
You can reasonably assume that all Dictionary tables that describe indexes have names containing the substring 'IND'. So:

```
SELECT * FROM DICT WHERE table_name LIKE 'USER%IND%';
```

Now you can see that the table you want is `USER_IND_COLUMNS`.

# Managing Procedures and Functions

# Handled Exceptions





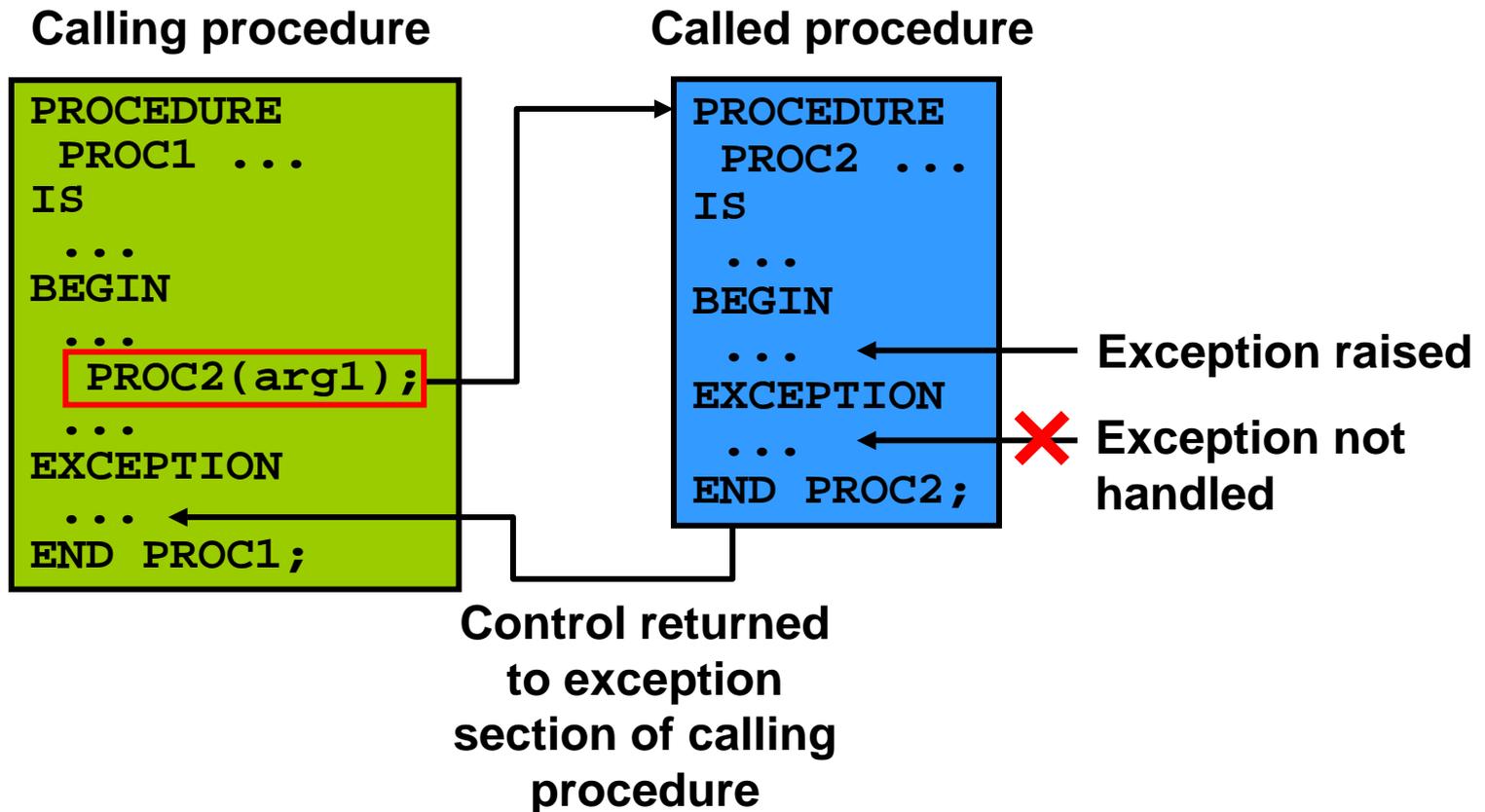
## Handled Exceptions: Example

```
CREATE OR REPLACE PROCEDURE add_department(  
    p_name VARCHAR2, p_mgr NUMBER, p_loc NUMBER) IS  
BEGIN  
    INSERT INTO DEPARTMENTS (department_id,  
        department_name, manager_id, location_id)  
    VALUES (DEPARTMENTS_SEQ.NEXTVAL, p_name, p_mgr, p_loc);  
    DBMS_OUTPUT.PUT_LINE('Added Dept: ' || p_name);  
EXCEPTION  
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE('Err: adding dept: ' || p_name);  
END;
```

```
CREATE PROCEDURE create_departments IS  
BEGIN  
    add_department('Media', 100, 1800);  
    add_department('Editing', 99, 1800);  
    add_department('Advertising', 101, 1800);  
END;
```



# Exceptions Not Handled





## Exceptions Not Handled: Example

```
CREATE OR REPLACE PROCEDURE add_department_noex(  
    p_name VARCHAR2, p_mgr NUMBER, p_loc NUMBER) IS  
BEGIN  
    INSERT INTO DEPARTMENTS (department_id,  
        department_name, manager_id, location_id)  
    VALUES (DEPARTMENTS_SEQ.NEXTVAL, p_name, p_mgr, p_loc);  
    DBMS_OUTPUT.PUT_LINE('Added Dept: ' || p_name);  
END;
```

```
CREATE OR REPLACE PROCEDURE create_departments_noex IS  
BEGIN  
    add_department_noex('Media', 100, 1800);  
    add_department_noex('Editing', 99, 1800);  
    add_department_noex('Advertising', 101, 1800);  
END;
```



## Removing Procedures and Functions

You can remove a procedure or function that is stored in the database.

- Syntax:

```
DROP {PROCEDURE procedure_name | FUNCTION function_name}
```

- Examples:

```
DROP PROCEDURE raise_salary;
```

```
DROP FUNCTION get_sal;
```

## Viewing Procedures/Functions in the Data Dictionary

The source code for PL/SQL subprograms is stored in the data dictionary tables. The source code is stored in the dictionary even when the PL/SQL subprogram did not compile successfully.

- The `USER_OBJECTS` table contains the names and types of procedures and functions.
- The `USER_SOURCE` table contains source code for all of the subprograms that you own.
- The `ALL_SOURCE` table contains source code for all the subprograms that you have privileges to invoke.



## Viewing Object Names in the USER\_OBJECTS Table

When you create any object, such as table, procedure, function, and so on, the entries are made to the USER\_OBJECTS table.

When the code in the slide is executed successfully, you can view the names of **functions/procedures** in the USER\_OBJECTS table by issuing the following command:

```
SELECT object_name
FROM    USER_OBJECTS
WHERE   object_type IN ( 'FUNCTION' , 'PROCEDURE' ) ;
```



## Viewing Source Code in the USER\_SOURCE Table

The source of the functions/procedures owned by your user name is stored in the USER\_SOURCE table. You can check the source for any function/procedure by issuing the following command:

```
SELECT text
FROM   USER_SOURCE
WHERE  type = 'FUNCTION' and name = 'TAX_FUNC'
ORDER BY line;
```

# Review of Object Privileges



## What Is an Object Privilege?

An object privilege allows the use of a specific database object, such as a table, a view or a PL/SQL procedure, by one or more database users.

When a database object is first created, only its owner (creator) and the Database Administrator are privileged to use it.

Privileges for all other users must be specifically granted (and can be later revoked). This can be done by the object owner or by the DBA.



## What Object Privileges Are Available?

Each object has a particular set of grantable privileges. The following table lists the privileges for various objects.

Object Privilege	Table	View	Sequence	Procedure
ALTER	X		X	
DELETE	X	X		
EXECUTE				X
INDEX	X			
INSERT	X	X		
REFERENCES	X	X		
SELECT	X	X	X	
UPDATE	X	X		

SELECT, INSERT, UPDATE, and DELETE privileges allow the holder (the grantee) of the privilege to use the corresponding SQL statement on the object.

For example, INSERT privilege on the EMPLOYEES table allows the holder to INSERT rows into the table, but not to UPDATE or DELETE rows.

 **What Object Privileges Are Available? (continued)**

Object Privilege	Table	View	Sequence	Procedure
ALTER	X		X	
DELETE	X	X		
EXECUTE				X
INDEX	X			
INSERT	X	X		
REFERENCES	X	X		
SELECT	X	X	X	
UPDATE	X	X		

ALTER privilege allows the grantee to ALTER the table, while INDEX privilege allows the grantee to create indexes on the table. Of course, you can automatically do this on your own tables!

REFERENCES privilege allows the grantee to check for the existence of rows in a table using foreign key constraints.

## Granting Object Privileges

Use the following syntax:

```
GRANT object_priv [(columns)]
ON object
TO {user/role|PUBLIC}
[WITH GRANT OPTION];
```

Examples:

```
GRANT INSERT, UPDATE ON employees TO TOM, SUSAN;

GRANT SELECT ON departments TO PUBLIC;
```

Syntax	Defined
object_priv	Is an object privilege to be granted
columns	Specifies a column from a table or view on which privileges are granted
ON object	Is the object on which the privileges are granted
user role	Identifies the user or role to whom the privilege is granted
PUBLIC	Grants object privileges to all users
WITH GRANT OPTION	Allows the grantee to grant the object privileges to other users and roles



## Revoking Object Privileges

Use the following syntax:

```
REVOKE object_priv [(columns)]  
ON object  
FROM {user/role | PUBLIC};
```

Examples:

```
REVOKE INSERT, UPDATE ON employees FROM TOM, SUSAN;  
  
REVOKE SELECT ON departments FROM PUBLIC;
```



## Using the EXECUTE privilege With Stored Subprograms

To invoke and execute a PL/SQL subprogram, the user must be granted EXECUTE privilege on the subprogram.

Example:

```
CREATE OR REPLACE PROCEDURE add_dept ... ;
CREATE OR REPLACE FUNCTION get_sal ...;

GRANT EXECUTE ON add_dept TO TOM, SUSAN;
GRANT EXECUTE ON get_sal TO PUBLIC;
...
REVOKE EXECUTE ON get_sal FROM PUBLIC;
```

## What About the Objects Referenced Inside the Subprogram?

To invoke a subprogram, a user needs only `EXECUTE` privilege on the subprogram. He/she does NOT need any privileges on the objects referenced by SQL statements within the subprogram.

```
CREATE OR REPLACE PROCEDURE add_dept ...
IS BEGIN
...
    INSERT INTO DEPARTMENTS ... ;
...
END;

GRANT EXECUTE ON add_dept TO SUSAN;
```

The user (SUSAN) does not need `INSERT` (or any other privilege) on the `DEPARTMENTS` table.

## So Who Needs Privileges on the Referenced Objects?

The subprogram owner (creator) must hold the appropriate privileges on the objects referenced by the subprogram. The owner's privileges are checked when the subprogram is created or replaced, and also every time the subprogram is invoked.

In this example, Tom creates a procedure that SUSAN needs to use:

```
(Table owner): GRANT INSERT ON departments TO TOM;
(Tom): CREATE OR REPLACE PROCEDURE add_dept ...
      IS BEGIN
          ...
          INSERT INTO DEPARTMENTS ... ;
          ...
      END;
(Tom): GRANT EXECUTE ON add_dept TO SUSAN;
```

# Using Invoker's Rights



## Definer's Rights Compared to Invoker's Rights

When a subprogram is called you might want the code to run with the invoker's set of privileges.

### Definer's rights:

- Programs execute with the privileges of the owner.
- User does not require privileges on underlying objects that the subprogram accesses. User only requires privilege to `EXECUTE` a subprogram.
- `DEFINER=OWNER`

### Invoker's rights:

- Programs execute with the privileges of the calling user.
- User requires privileges on the underlying objects that the subprogram accesses.
- `INVOKER=USER`

 **Example**

By setting AUTHID to CURRENT\_USER, anyone calling this function requires privileges on the referenced database objects. In this case, SELECT on the tests table.

```
CREATE OR REPLACE PROCEDURE grades
  (p_score OUT NUMBER, p_name IN VARCHAR2)
  AUTHID CURRENT_USER
IS
BEGIN
  SELECT score INTO p_score
  FROM tests
  WHERE key=p_name;
END;
```



## Another Example

```
Tom> CREATE OR REPLACE PROCEDURE add_dept ...  
      [AUTHID CURRENT_USER] IS  
      BEGIN  
          INSERT INTO departments ...;  
      END;  
Tom> GRANT EXECUTE ON add_dept TO susan;  
  
Susan> BEGIN    tom.add_dept(...); END;
```

If Definer's Rights (the default) are used, Tom needs INSERT privilege on DEPARTMENTS. If Invoker's Rights (with AUTHID CURRENT\_USER) are used, SUSAN needs INSERT privilege on DEPARTMENTS.

In both cases, SUSAN needs EXECUTE privilege on ADD\_DEPT.

## In Which Schema Are Referenced Objects?

Using Definer's Rights, objects are assumed to be in the definer's schema:

```
Tom> CREATE TABLE tests ... ;
Tom> CREATE OR REPLACE PROCEDURE grades ... IS
    BEGIN
        ... SELECT ... FROM tests ... ;
    END;

Susan> BEGIN ... tom.grades(...); END;
```

When using Definer's Rights, table-name TESTS is resolved in the definer's schema. This means that table TESTS is assumed to be in Tom's schema, so you do not need to prefix the table-name with the schema-name. This procedure compiles and executes successfully provided the relevant privileges have been granted.

## In Which schema Are Referenced Objects? (continued)

When using Invoker's Rights, referenced objects are assumed to be in the invoker's schema:

```
Tom> CREATE TABLE tests ... ;
Tom> CREATE OR REPLACE PROCEDURE grades ...
    AUTHID CURRENT_USER IS
    BEGIN
    ... SELECT ... FROM tests ... ; -- Line A
    END;

Susan> BEGIN ... tom.grades(...); END;
```

Using Invoker's rights, object-names are resolved in the Invoker's schema. This means that TESTS is now assumed to be in Susan's schema. If TESTS is in Tom's schema, Line A must be changed to:

```
... SELECT ... FROM tom.tests ... ; -- Line A
```