Design consideration for PL/SQL Code

Standardizing Constants and Exceptions

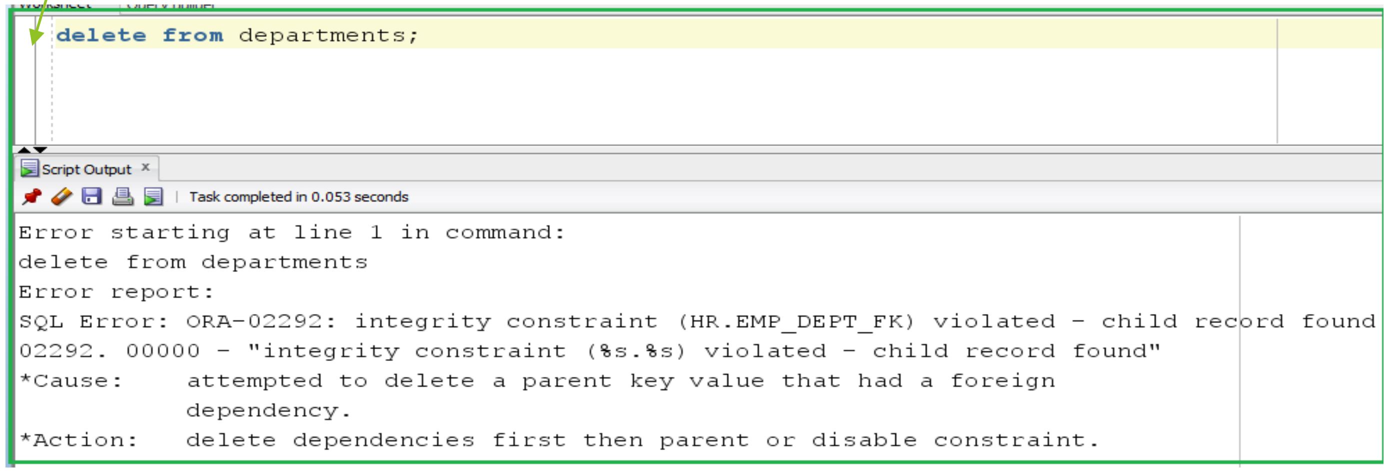
Constants and exceptions are typically implemented using a bodiless package (that is, a package specification).

- Standardizing helps to:
  - Develop programs that are consistent
  - Promote a higher degree of code reuse
  - Ease code maintenance
  - Implement company standards across entire applications
- Start with standardization of:
  - Exception names
  - Constant definitions

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
  e_fk_err      EXCEPTION;
  e_seq_nbr_err EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_fk_err, -2292);
  PRAGMA EXCEPTION_INIT (e_seq_nbr_err, -2277);
  ...
END error_pkg;
/
```



```
delete from departments;
```

Script Output ×

Task completed in 0.053 seconds

```
Error starting at line 1 in command:
delete from departments
Error report:
SQL Error: ORA-02292: integrity constraint (HR.EMP_DEPT_FK) violated - child record found
02292. 00000 - "integrity constraint (%s.%s) violated - child record found"
*Cause:    attempted to delete a parent key value that had a foreign
           dependency.
*Action:   delete dependencies first then parent or disable constraint.
```

```
...
Begin
Delete form departments where
Department_id=p_dept_id
...
...
Exception
when error_pkg.e_fk_err then...
....
End;
```

Standardizing Exception Handling

Consider writing a subprogram for common exception handling to:

- Display errors based on `SQLCODE` and `SQLERRM` values for exceptions
- Track run-time errors easily by using parameters in your code to identify:
  - The procedure in which the error occurred
  - The location (line number) of the error
  - `RAISE_APPLICATION_ERROR` using stack trace capabilities, with the third argument set to `TRUE`

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
create or replace package global_Measurement
is

  c_mile_to_km    constant number:=1.6093;
  c_kilo_to_mile constant number:=0.6214;

end;
```

Local Subprograms

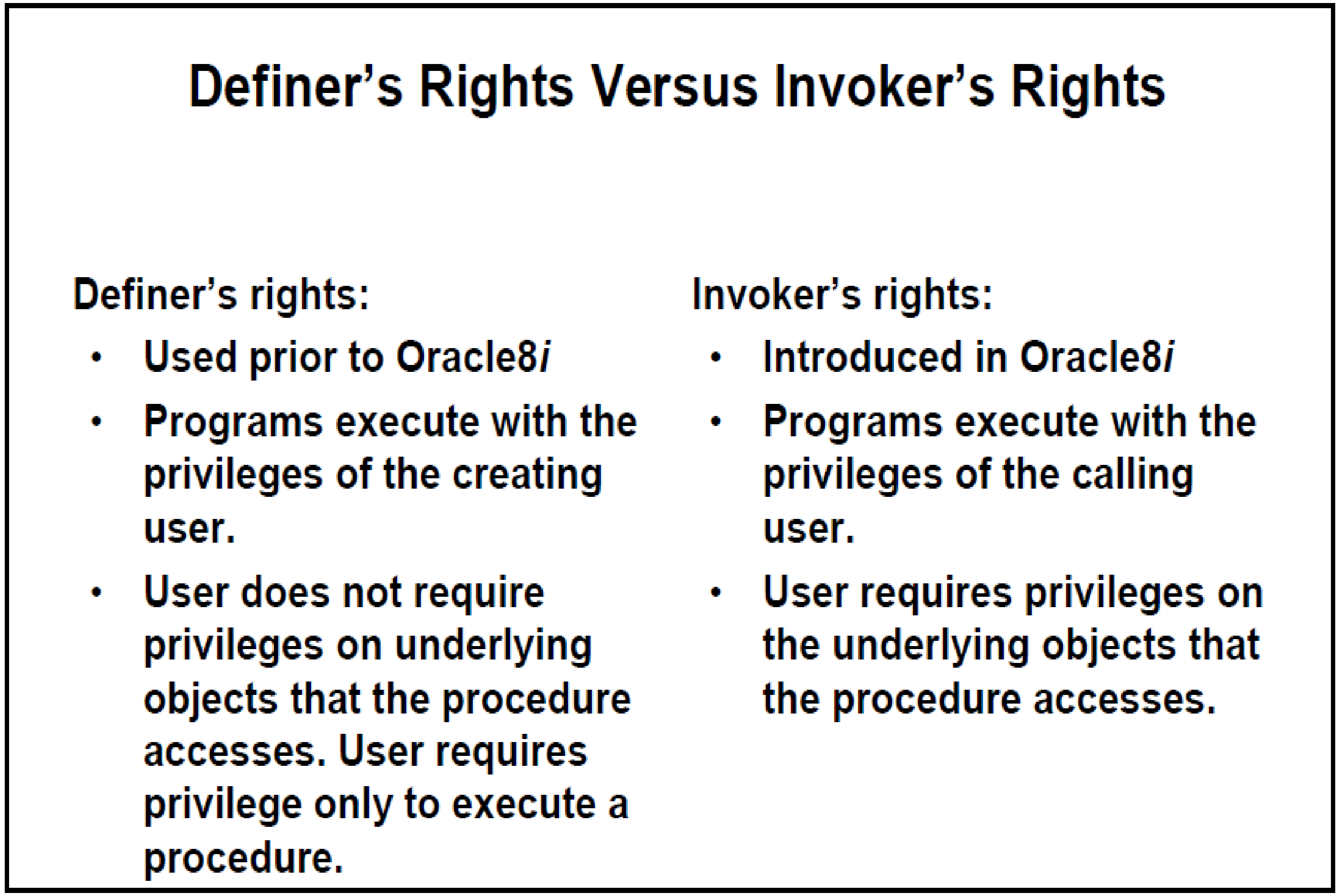
- A local subprogram is a `PROCEDURE` or `FUNCTION` defined at the end of the declarative section.

```
CREATE PROCEDURE employee_sal(p_id NUMBER) IS
   v_emp employees%ROWTYPE;
   FUNCTION tax(p_salary VARCHAR2) RETURN NUMBER IS
   BEGIN
     RETURN p_salary * 0.825;
   END tax;
BEGIN
   SELECT * INTO v_emp
   FROM EMPLOYEES WHERE employee_id = p_id;
   DBMS_OUTPUT.PUT_LINE('Tax: '||tax(v_emp.salary));
END;
/
EXECUTE employee_sal(100)
```

Local Function

1- can be accessible only from the block owner
2- compiled as a part of the owner block

Why using local subprogram?
1- reduction of repetitive code
2- code readability
3- easy maintenance



Definer's Rights Versus Invoker's Rights

Definer's rights:

- Used prior to Oracle8*i*
- Programs execute with the privileges of the creating user.
- User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.

Invoker's rights:

- Introduced in Oracle8*i*
- Programs execute with the privileges of the calling user.
- User requires privileges on the underlying objects that the procedure accesses.

Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER

The default is AUTHID DEFINER.

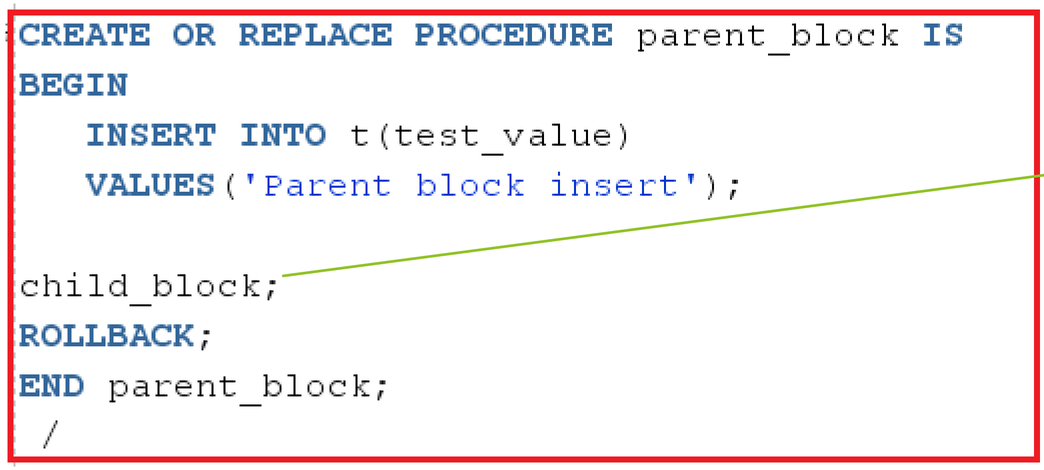
```
CREATE OR REPLACE PROCEDURE add_dept(
  p_id NUMBER, p_name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
  INSERT INTO departments
  VALUES (p_id, p_name, NULL, NULL);
END;
```

When used with stand-alone functions, procedures, or packages:

- Names used in queries, DML, Native Dynamic SQL, and DBMS_SQL package are resolved in the invoker's schema
- Calls to other packages, functions, and procedures are resolved in the definer's schema

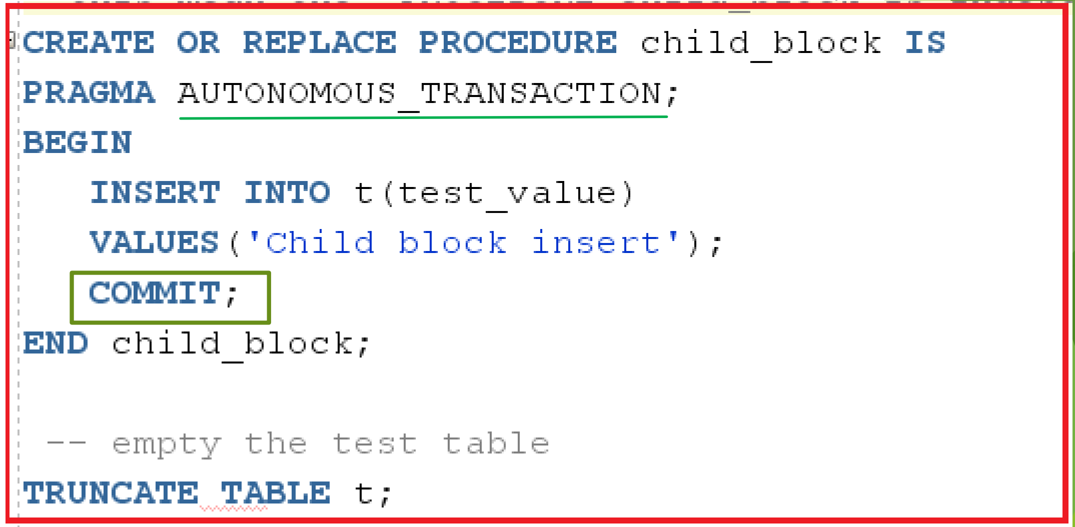
Autonomous Transactions

- Are independent transactions started by another main transaction
- Are specified with `PRAGMA AUTONOMOUS_TRANSACTION`



```
CREATE OR REPLACE PROCEDURE parent_block IS
BEGIN
   INSERT INTO t(test_value)
   VALUES('Parent block insert');

child_block;
ROLLBACK;
END parent_block;
 /
```



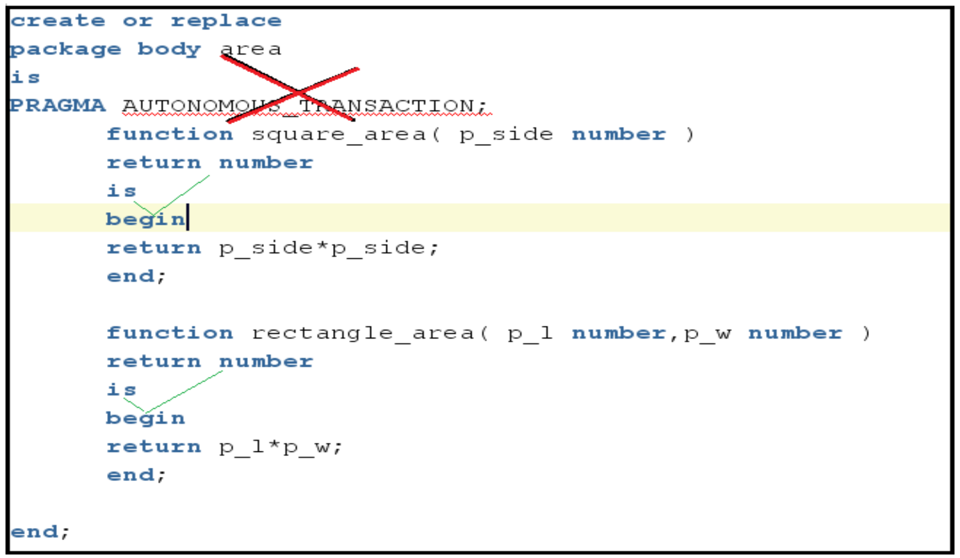
```
CREATE OR REPLACE PROCEDURE child_block IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO t(test_value)
   VALUES('Child block insert');
   COMMIT;
END child_block;

 -- empty the test table
TRUNCATE TABLE t;
```

This mean this procedure is independent
The commit not affect the caller procedure

Features of Autonomous Transactions

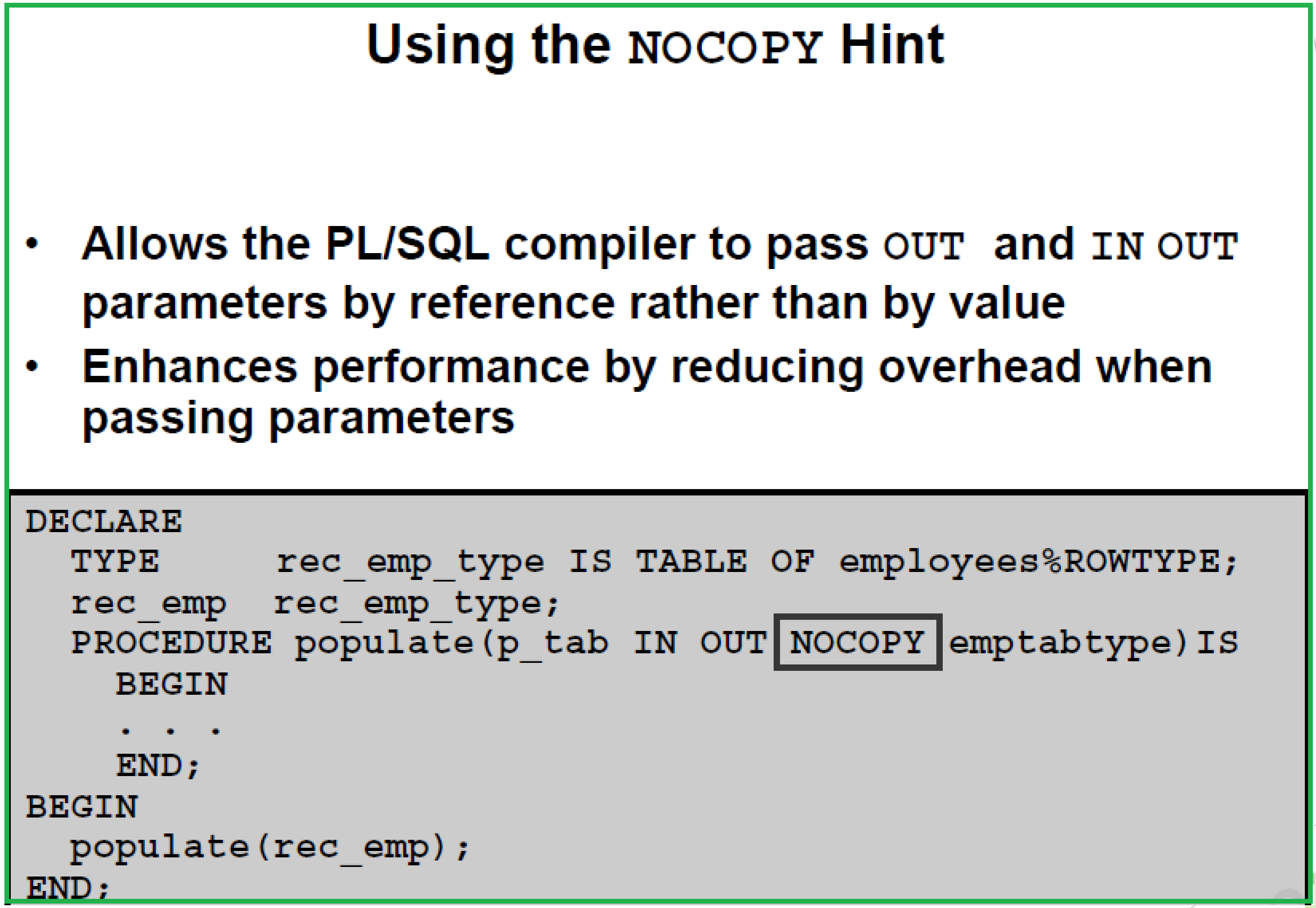
- Are independent of the main transaction
- Suspend the calling transaction until the autonomous transactions are completed
- Are not nested transactions
- Do not roll back if the main transaction rolls back
- Enable the changes to become visible to other transactions upon a commit
- Are started and ended by individual subprograms and not by nested or anonymous PL/SQL blocks



```
create or replace
package body area
is
PRAGMA AUTONOMOUS_TRANSACTION;
      function square_area( p_side number )
      return number
      is
      begin
      return p_side*p_side;
      end;

      function rectangle_area( p_l number,p_w number )
      return number
      is
      begin
      return p_l*p_w;
      end;

end;
```



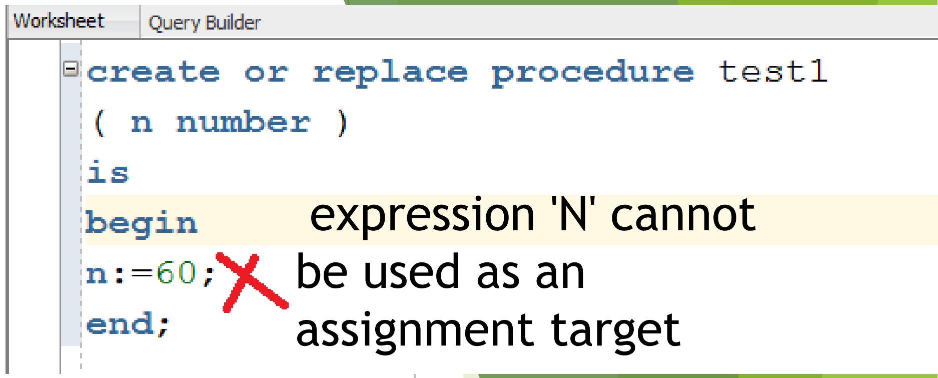
Using the NOCOPY Hint

- Allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value
- Enhances performance by reducing overhead when passing parameters

```
DECLARE
  TYPE     rec_emp_type IS TABLE OF employees%ROWTYPE;
  rec_emp  rec_emp_type;
  PROCEDURE populate(p_tab IN OUT NOCOPY emptabtype)IS
    BEGIN
    . . .
    END;
BEGIN
  populate(rec_emp);
END;
```

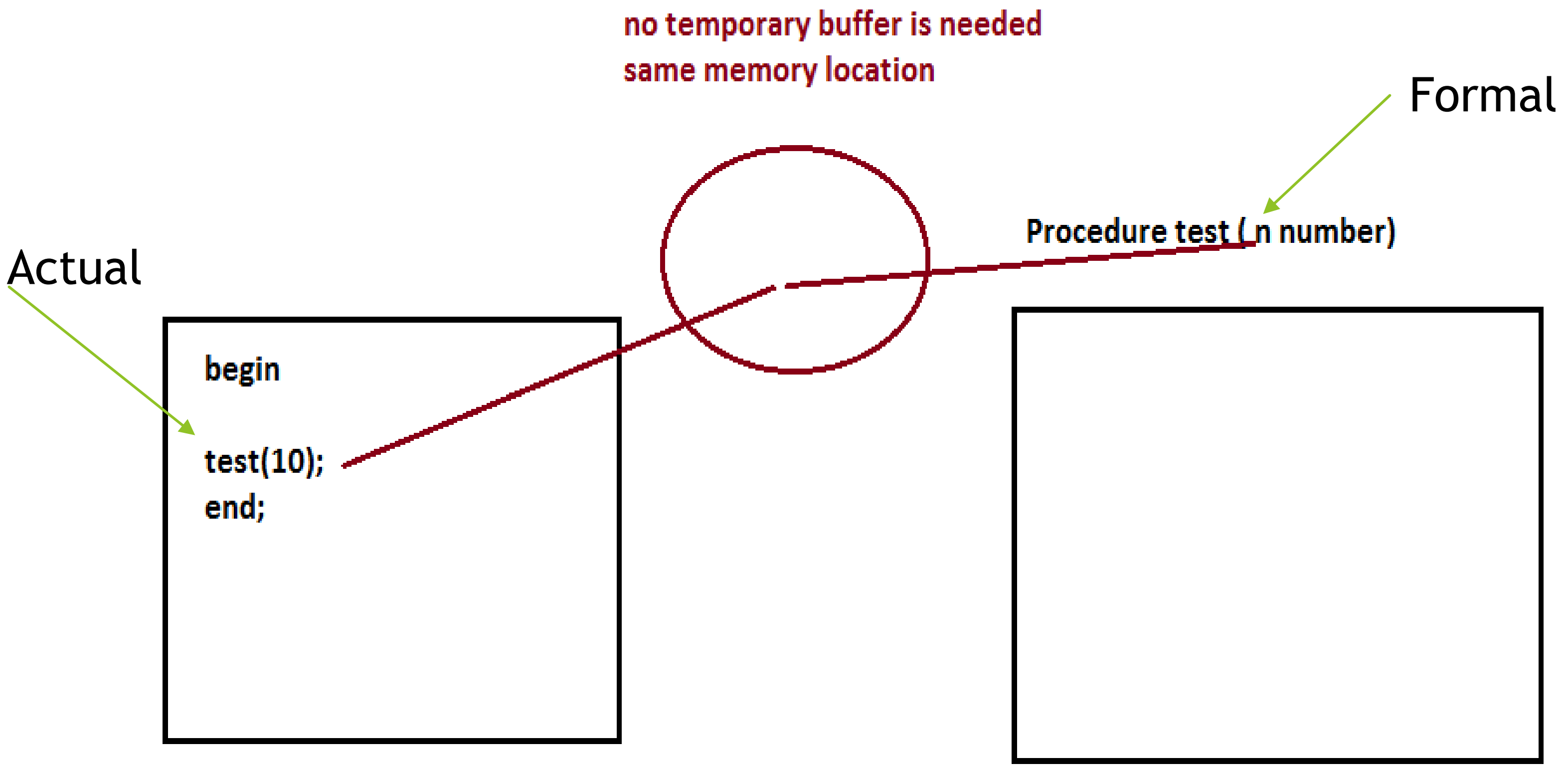
We use NOCOPY in complexe data types (LOBs, XMLTYPEs, collections etc.)

IN parameter always passed by reference



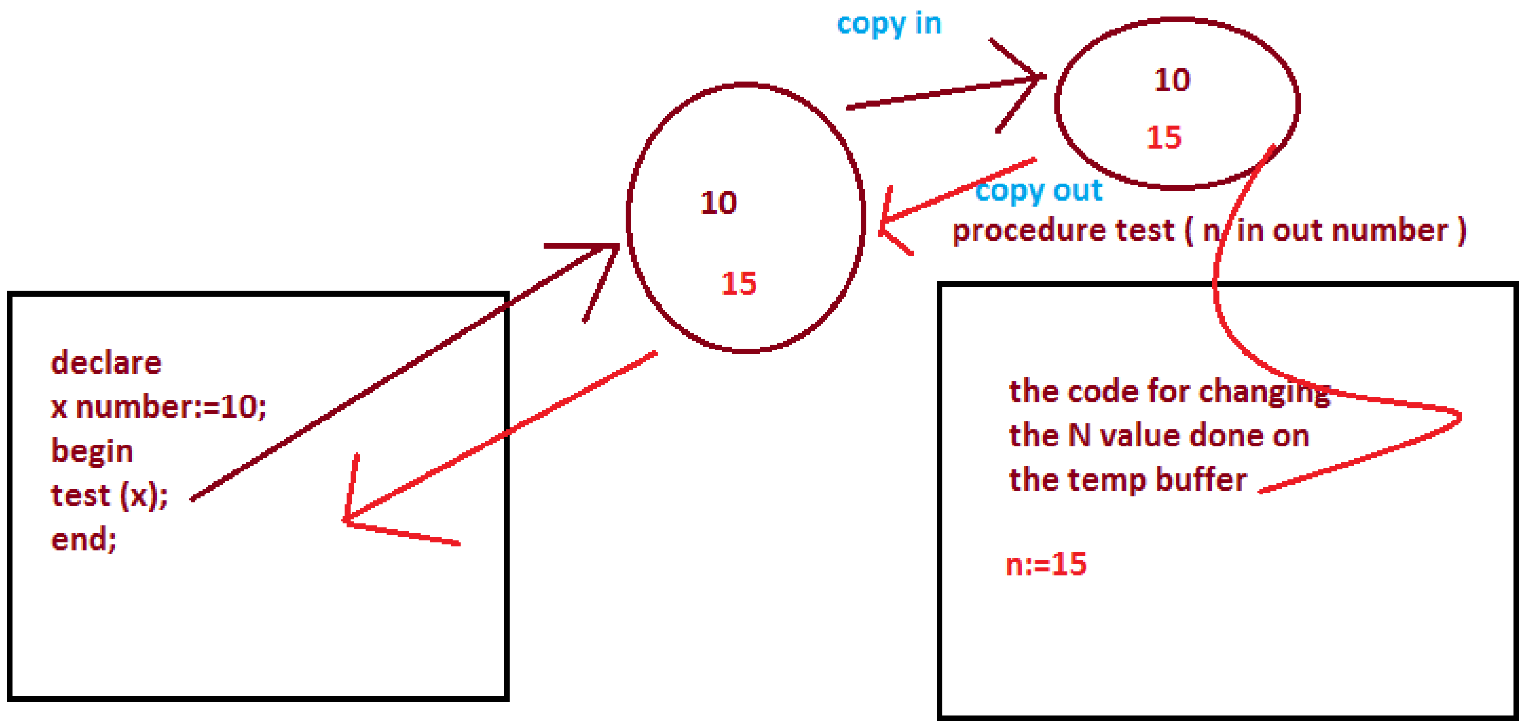
```
create or replace procedure test1
( n number )
is
begin
n:=60;
end;
```

expression 'N' cannot be used as an assignment target



OUT/IN OUT parameters can be passed
1-Pass By Value (default)
2-Pass By Reference using no copy

1-Pass By Value (default)



```
declare
x number:=10;
begin
test (x);
end;
```

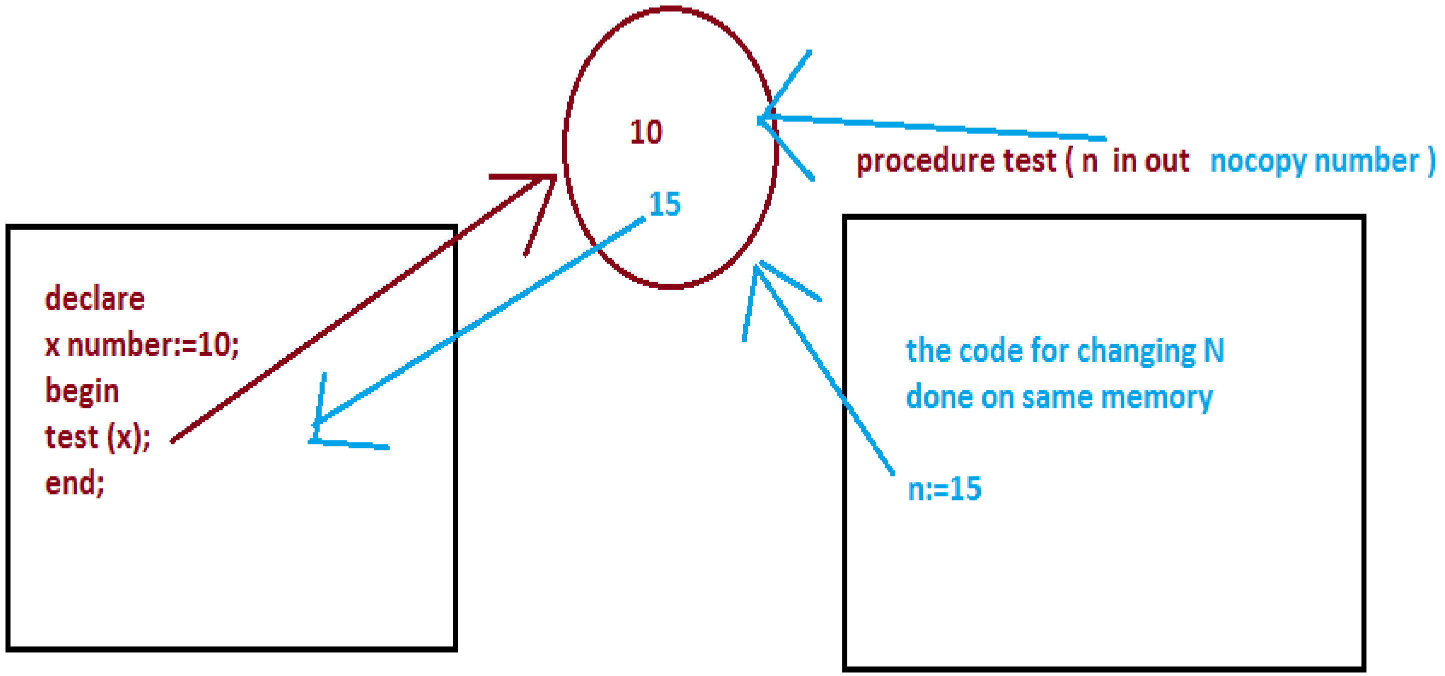
```
procedure test ( n in out number )
```

the code for changing the N value done on the temp buffer

n:=15

OUT/IN OUT parameters can be passed
1-Pass By Value (default)
2-Pass By Reference using nocopy

2-Pass By Reference using nocopy

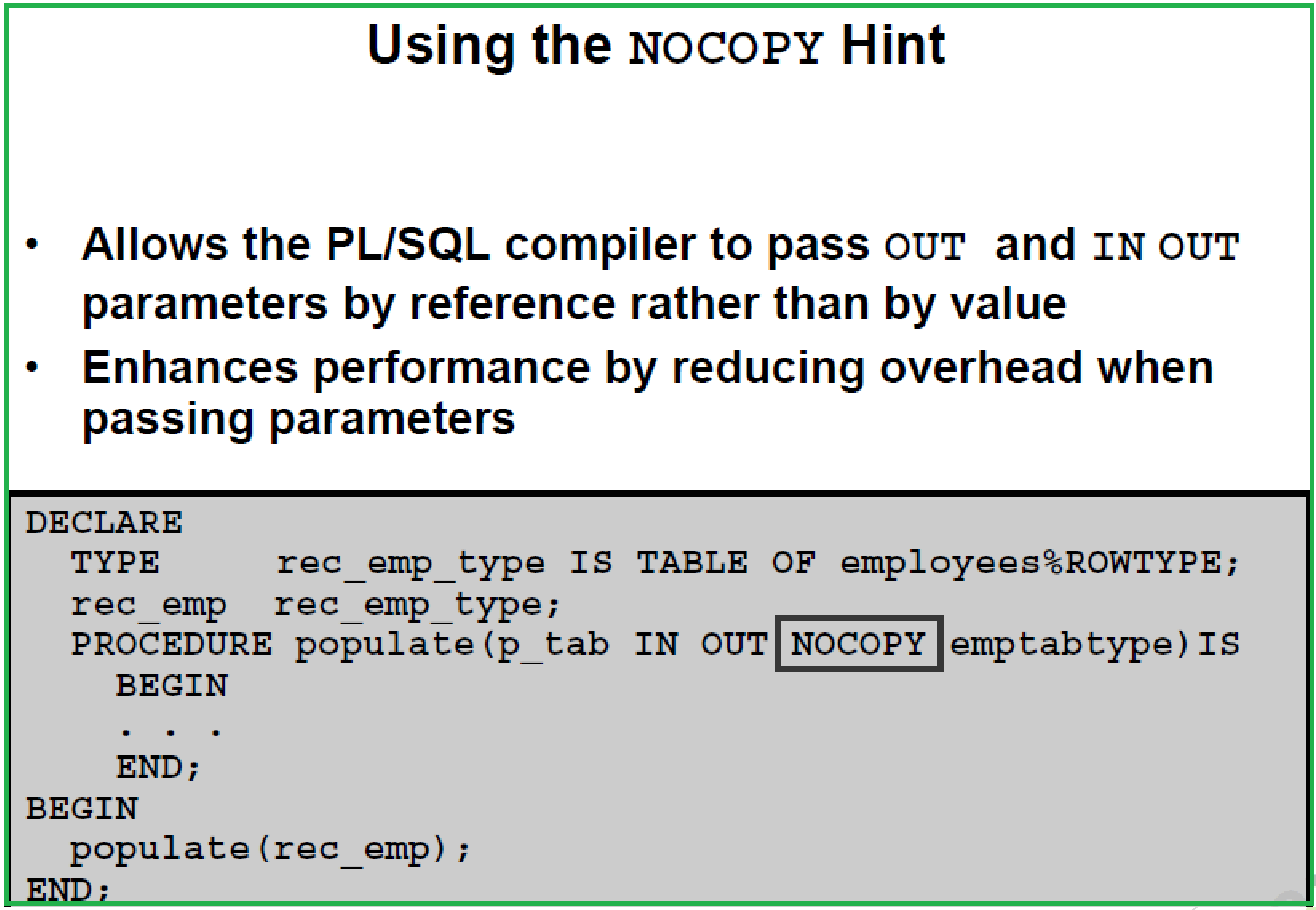


```
declare
x number:=10;
begin
test (x);
end;
```

```
procedure test ( n  in out   nocopy number )
```

the code for changing N
done on same memory

```
n:=15
```



Using the NOCOPY Hint

- Allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value
- Enhances performance by reducing overhead when passing parameters

```
DECLARE
  TYPE      rec_emp_type IS TABLE OF employees%ROWTYPE;
  rec_emp   rec_emp_type;
  PROCEDURE populate(p_tab IN OUT NOCOPY emptabtype)IS
    BEGIN
    . . .
    END;
BEGIN
  populate(rec_emp);
END;
```

We use NOCOPY in complex data types (LOBs, XMLTYPEs, collections etc.)

Effects of the NOCOPY Hint

- If the subprogram exits with an exception that is not handled:
  - You cannot rely on the values of the actual parameters passed to a NOCOPY parameter
  - Any incomplete modifications are not “rolled back”
- The remote procedure call (RPC) protocol enables you to pass parameters only by value.

When Does the PL/SQL Compiler Ignore the NOCOPY Hint?

The NOCOPY hint has no effect if:

- The actual parameter:
  - Is an element of an index-by table
  - Is constrained (for example, by scale or NOT NULL)   number
  - And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ
  - Requires an implicit data type conversion
- The subprogram is involved in an external or remote procedure call

Using the `PARALLEL_ENABLE` Hint

- Can be used in functions as an optimization hint
- Indicates that a function can be used in a parallelized query or parallelized DML statement

```
CREATE OR REPLACE FUNCTION f2 (p_p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p_p1 * 2;
END f2;
```

```
SELECT /*+ PARALLEL(4) */
First_name, f2(employee_id)
From
employees
```

Oracle will open 4 processes to execute this statement
Each process take subset of data

Cache and parallel are only in oracle enterprise editions
https://docs.oracle.com/database/121/DBLIC/editions.htm#DBLIC116

Using the Cross-Session PL/SQL Function Result Cache

- Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in cache.
- The function result cache is stored in a shared global area (SGA), making it available to any session that runs your application.
- Subsequent calls to the same function with the same parameters uses the result from cache.
- Improves performance and scalability.
- Use with functions that are called frequently and dependent on information that changes infrequently.

Note: if the database object that used to compute the value changed , then result recomputed

Note: If function execution results in an unhandled exception, the exception result is not stored in the cache.

Enabling Result-Caching for a Function

available since Oracle 11g

You can make a function result-cached as follows:

- Include the `RESULT_CACHE` clause in the following:
  - The function declaration
  - The function definition
- Include an optional `RELIES_ON` clause, to specify any tables or views on which the function results depend.

```
CREATE OR REPLACE FUNCTION emp_hire_date (p_emp_id
    NUMBER) RETURN VARCHAR
 RESULT_CACHE RELIES_ON (employees) IS
  v_date_hired DATE;
BEGIN
  SELECT hire_date INTO v_date_hired
  FROM HR.Employees
  WHERE Employee_ID = p_emp_ID;
  RETURN to_char(v_date_hired);
END;
```

Cache and parallel are only in oracle enterprise editions
https://docs.oracle.com/database/121/DBLIC/editions.htm#DBLIC116

Using the `DETERMINISTIC` Clause with Functions

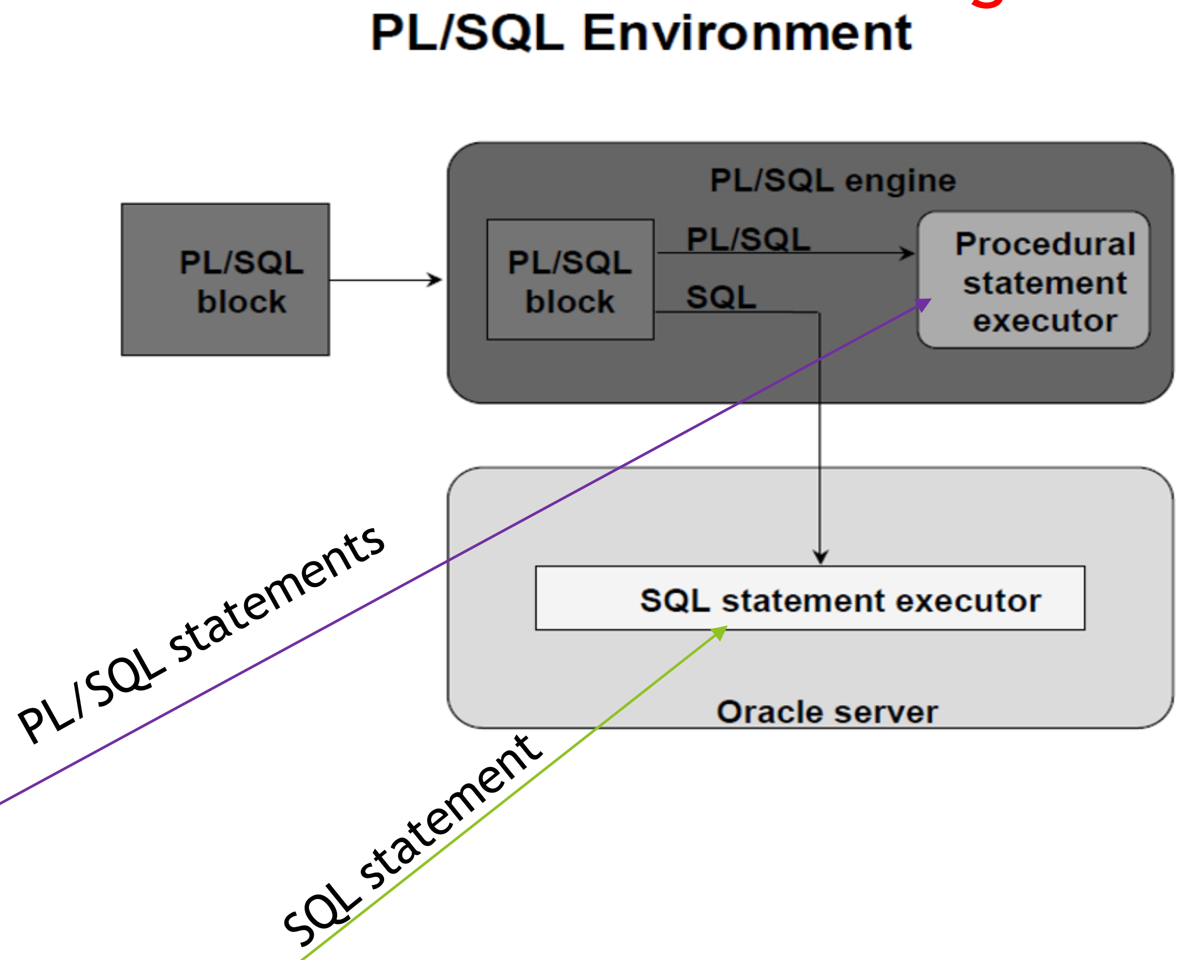
The DETERMINSTIC hint has been available since Oracle 8i

- Specify `DETERMINISTIC` to indicate that the function returns the same result value whenever it is called with the same values for its arguments.
- This helps the optimizer avoid redundant function calls.
- If a function was called previously with the same arguments, the optimizer can elect to use the previous result.
- Do not specify `DETERMINISTIC` for a function whose result depends on the state of session variables or schema objects.

If another session runs the same code with the same parameters the code is still executed. The cache is not shared between sessions.

You must specify this keyword if you intend to call the function in the expression of a function-based index or from the query of a materialized view that is marked REFRESH FAST or ENABLE QUERY REWRITE.
When Oracle Database encounters a deterministic function in one of these contexts, it attempts to use previously calculated results when possible rather than reexecuting the function. If you subsequently change the semantics of the function, you must manually rebuild all dependent function-based indexes and materialized views.

BULK Binding



```
Begin
…
..
For I in (select * from x )
Loop
Insert into….
End loop;
…
```

When dealing with collections this make the performance less
So Oracle introduce something called BULK Binding
(send the SQL statements using bulk)

Using Bulk Binding: Syntax and Keywords

- The `FORALL` keyword instructs the *PL/SQL engine* to bulk bind input collections before sending them to the SQL engine.

```
FORALL index IN lower_bound .. upper_bound
  [SAVE EXCEPTIONS]
  sql_statement;
```

- The `BULK COLLECT` keyword instructs the *SQL engine* to bulk bind output collections before returning them to the PL/SQL engine.

```
... BULK COLLECT INTO
      collection_name[,collection_name] ...
```

Handling FORALL Exceptions with the %BULK_EXCEPTIONS Attribute

To manage exceptions and have the bulk bind complete despite errors, add the keywords SAVE EXCEPTIONS to your FORALL statement after the bounds, before the DML statement.

All exceptions raised during the execution are saved in the cursor attribute %BULK_EXCEPTIONS, which stores a collection of records. Each record has two fields:

%BULK_EXCEPTIONS(I).ERROR_INDEX holds the "iteration" of the FORALL statement during which the exception was raised and %BULK_EXCEPTIONS(i).ERROR_CODE holds the corresponding Oracle error code.

Values stored in %BULK_EXCEPTIONS refer to the most recently executed FORALL statement. Its subscripts range from 1 to %BULK_EXCEPTIONS.COUNT.

Thank You