SQL Programming

CSCI 4127

Advanced Database

Outline

- DBMS Programming Languages
- SQL Programming
- SubPrograms

DB Procedural Languages

- Oracle - PL/SQL
- MS SQL Server - Transact-SQL or T-SQL
- MySQL - ANSI SQL
- PostgreSQL - Install your choice

Why?

- Decouple business logic from Middle Tier or GUI
- Maintain in one place
- Speed implications

Features

- More power than traditional SQL
- Conditional statements
- Loops
- Creation of temp tables
- Error Handling
- Case Insensitive

Advantages to DB Programming

- Tight integration with SQL
- High performance
  - Reduced network traffic
  - Compiled and stored on server
- OOP support
- Security
  - Code is moved from client to server
- Enforce business rules

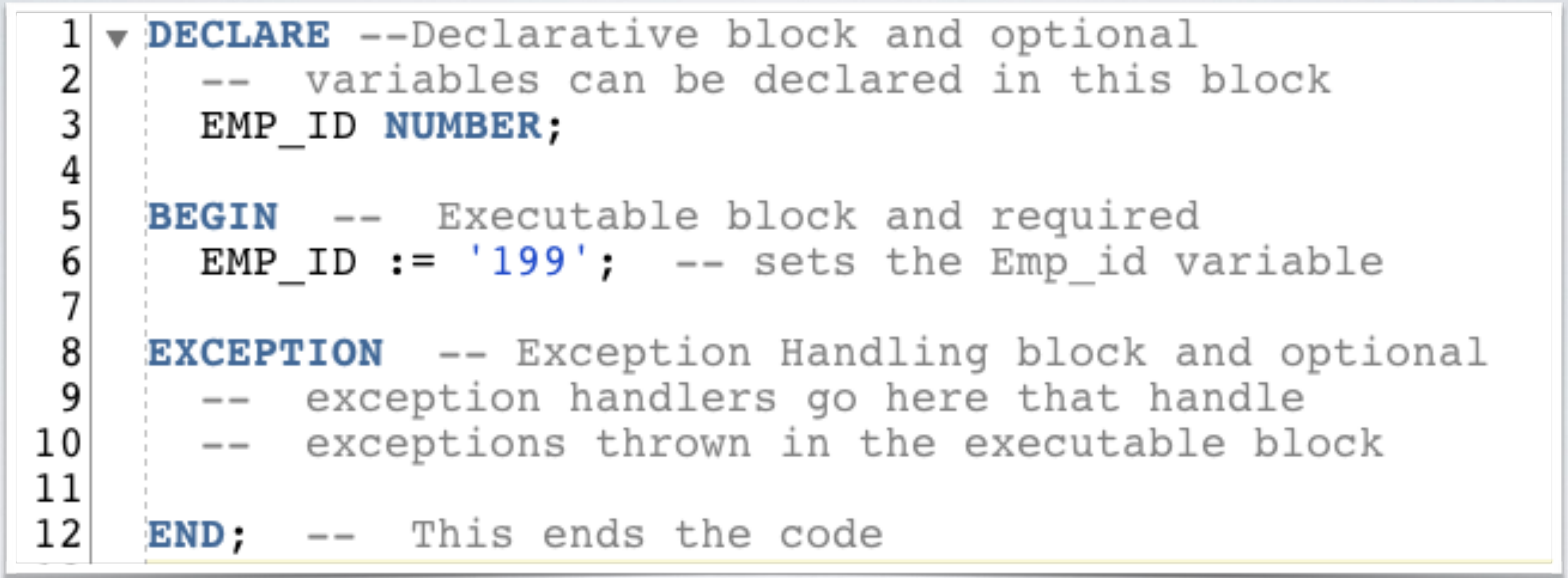
PL/SQL Features

- Blocks
- Error Handling
- I/O
- Variables and Constants
- Control Structures
- Cursors

Block

- Basic unit
- Consists of
  - [Declare]
  - Begin/End
  - [Exception]

Example

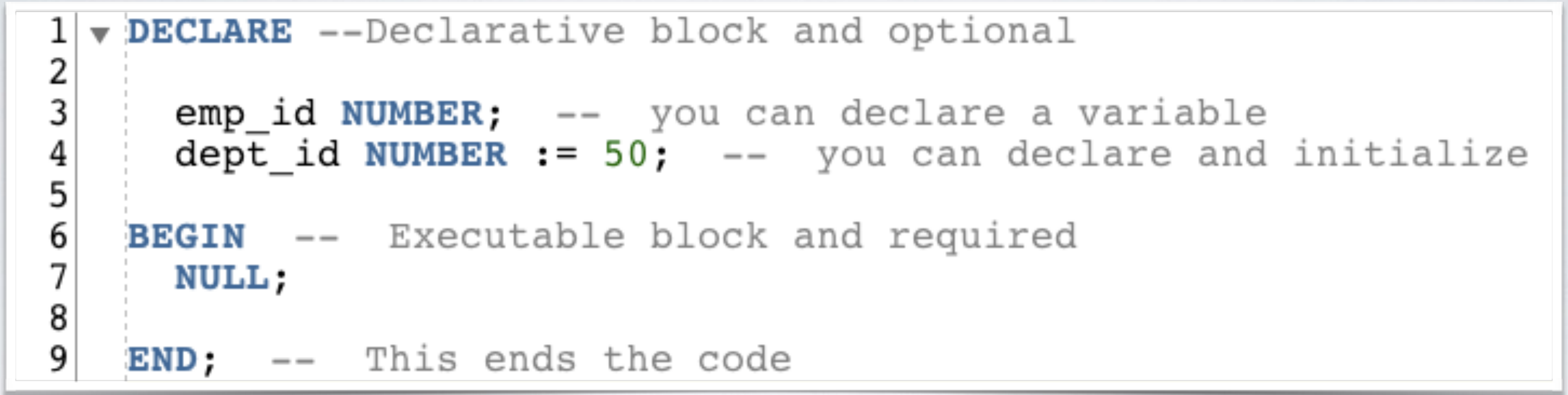


```
DECLARE --Declarative block and optional
  --  variables can be declared in this block
  EMP_ID NUMBER;

BEGIN  --  Executable block and required
  EMP_ID := '199';  -- sets the Emp_id variable

EXCEPTION  -- Exception Handling block and optional
  --  exception handlers go here that handle
  --  exceptions thrown in the executable block

END;  --  This ends the code
```

Declaring Variables

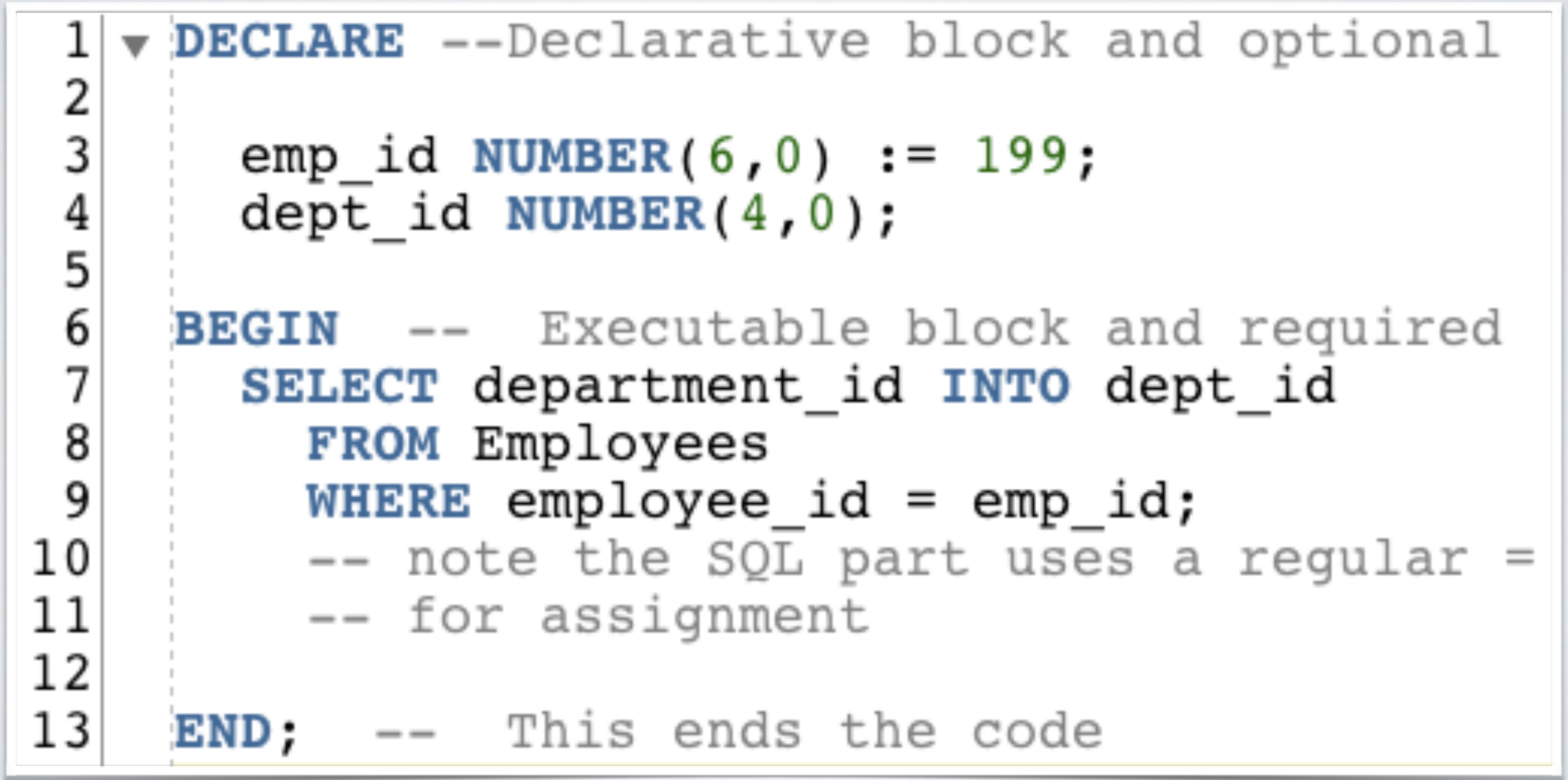
- Can have any SQL data type or PL/SQL Data type
  - BOOLEAN, PLS_INTEGER
  - Collections, Nested Tables
- Setting variables
  - Assignment (:= is assignment operator)
  - Selecting DB values into variables
  - Using subprogram parameters

Variable Assignment



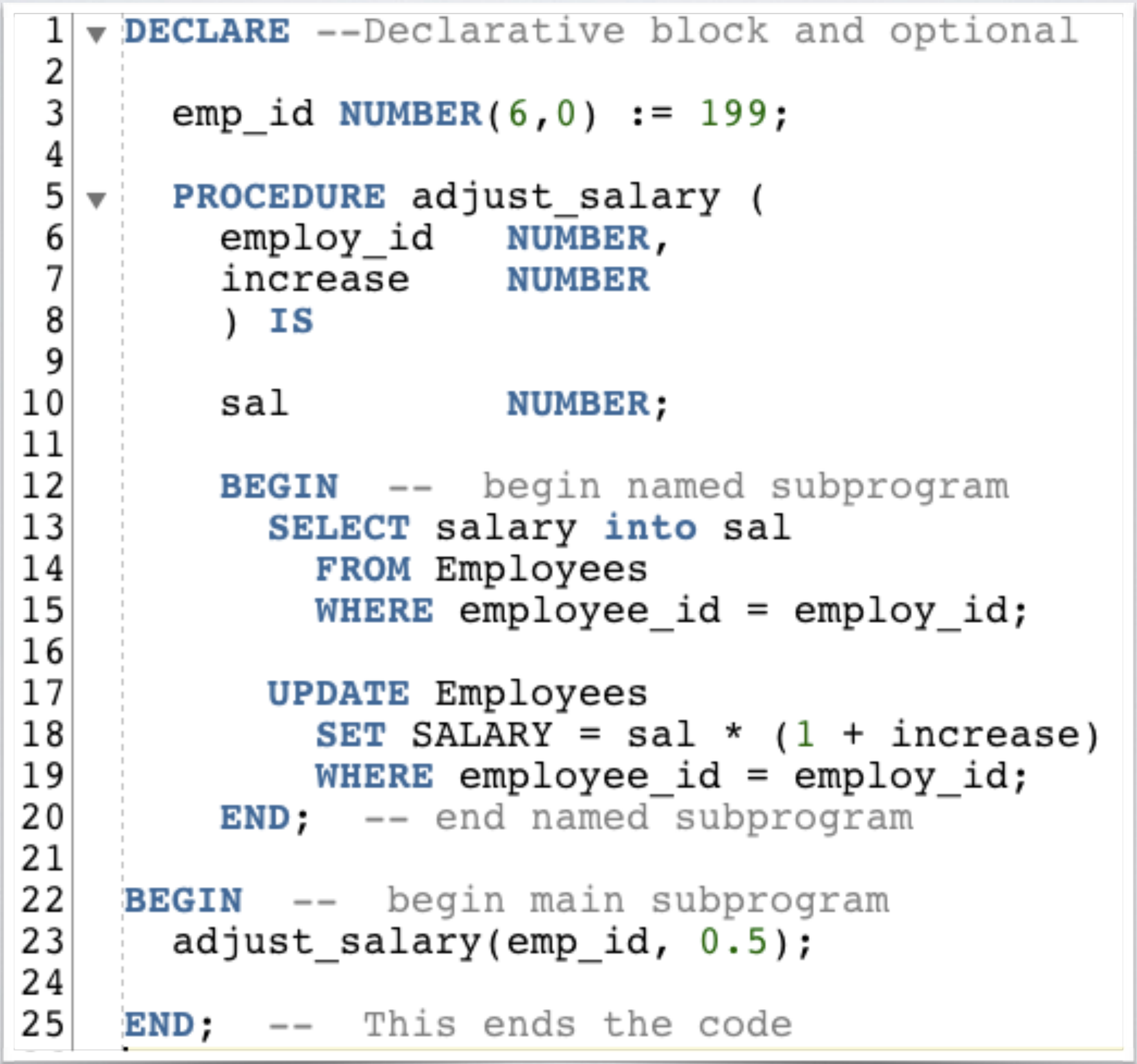
```
DECLARE --Declarative block and optional

  emp_id NUMBER;  --  you can declare a variable
  dept_id NUMBER := 50;  --  you can declare and initialize

BEGIN  --  Executable block and required
  NULL;

END;  --  This ends the code
```

Values From Database



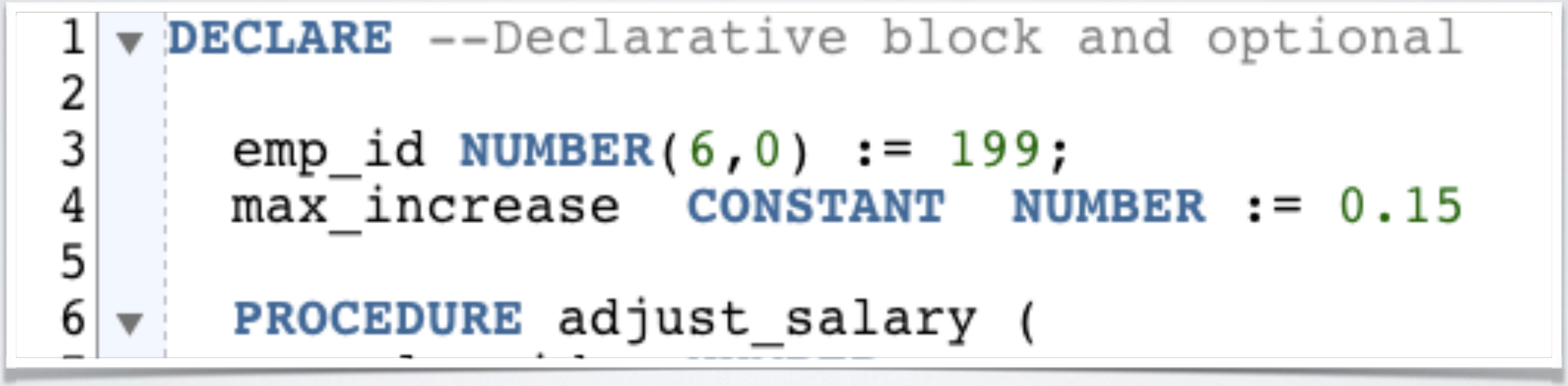
```
DECLARE --Declarative block and optional

  emp_id NUMBER(6,0) := 199;
  dept_id NUMBER(4,0);

BEGIN  --  Executable block and required
  SELECT department_id INTO dept_id
    FROM Employees
    WHERE employee_id = emp_id;
    -- note the SQL part uses a regular =
    -- for assignment

END;  --  This ends the code
```

Subprogram Parameters



```
DECLARE --Declarative block and optional

  emp_id NUMBER(6,0) := 199;

  PROCEDURE adjust_salary (
    employ_id   NUMBER,
    increase    NUMBER
    ) IS

    sal         NUMBER;

    BEGIN   --  begin named subprogram
      SELECT salary into sal
        FROM Employees
        WHERE employee_id = employ_id;

      UPDATE Employees
        SET SALARY = sal * (1 + increase)
        WHERE employee_id = employ_id;
    END;   -- end named subprogram

BEGIN   --  begin main subprogram
  adjust_salary(emp_id, 0.5);

END;   --  This ends the code
```

Constants



```
DECLARE --Declarative block and optional

  emp_id NUMBER(6,0) := 199;
  max_increase  CONSTANT  NUMBER := 0.15

  PROCEDURE adjust_salary (
```

Subprogram Parameter Modes

- In - Default mode
  - Value passed in but subprogram cannot change this value
- Out
  - Returns a value to the invoker
- In Out
  - Passes value to subprogram and returns value to invoker

Bind Variables

- Improve SQL performance through reuse
  - Reuse execution plan if stmt exactly the same
- Embedded Insert, Update, Delete, and Select SQL
  - Variable in WHERE and VALUES
- DBMS can reuse the SQL statements and sub in different values for bind variables
- Prevent SQL injection

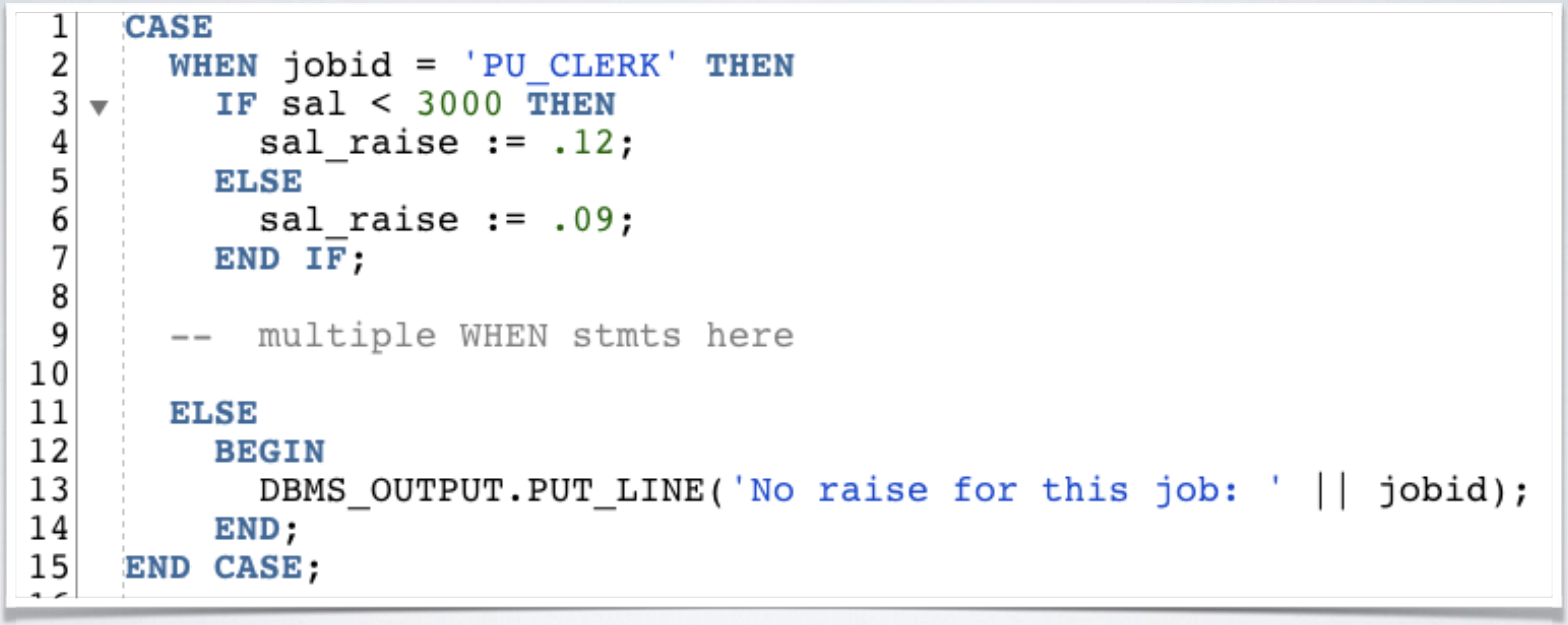
Special Variable Types

- %TYPE - variable that will match a column in the DB
    - v_last_name employees.last_name%TYPE;
- %ROWTYPE - variable that will match a row in a table or a result set
    - v_emp employees%ROWTYPE;

Control Statements

- Conditional
- Iterative
- Sequential - GOTO - we don’t talk about this. Look up spaghetti code.

Conditional Control

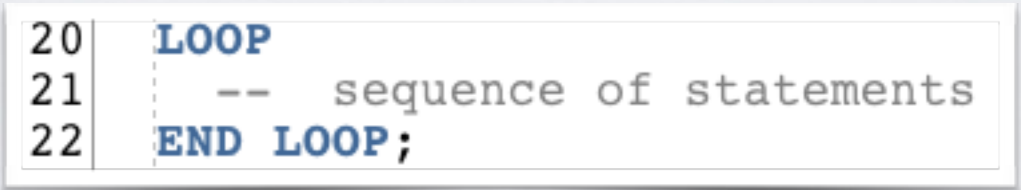


```
CASE
  WHEN jobid = 'PU_CLERK' THEN
    IF sal < 3000 THEN
      sal_raise := .12;
    ELSE
      sal_raise := .09;
    END IF;

  --  multiple WHEN stmts here

  ELSE
    BEGIN
      DBMS_OUTPUT.PUT_LINE('No raise for this job: ' || jobid);
    END;
END CASE;
```

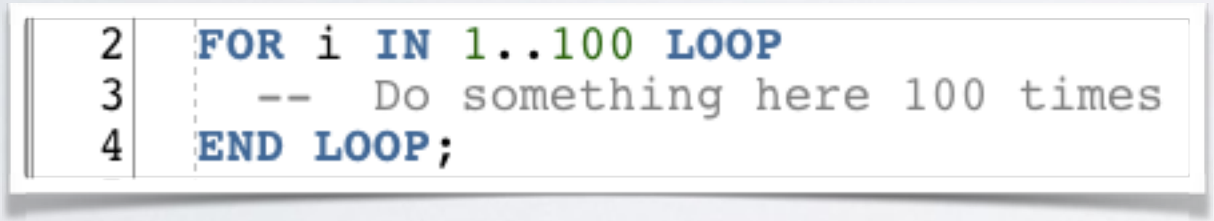
Iterative Control Statements

- LOOP - repeats a sequence of stmts continually
- FOR-LOOP - repeats a sequence of stmts a set integer number of times
- WHILE-LOOP - repeats a sequence of stmts while a condition is true. Sequence may never run.
- EXIT-WHEN - repeats a sequence of stmts until an exit condition is true. Sequence will run at least once.

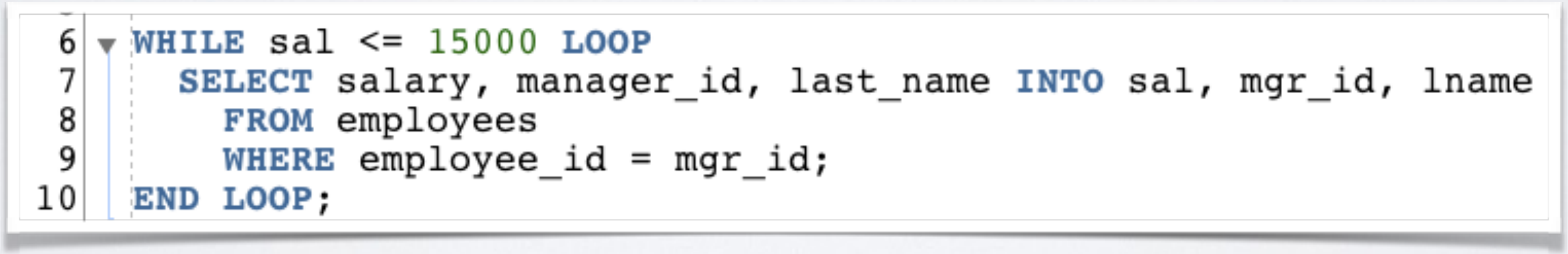
Iterative Control



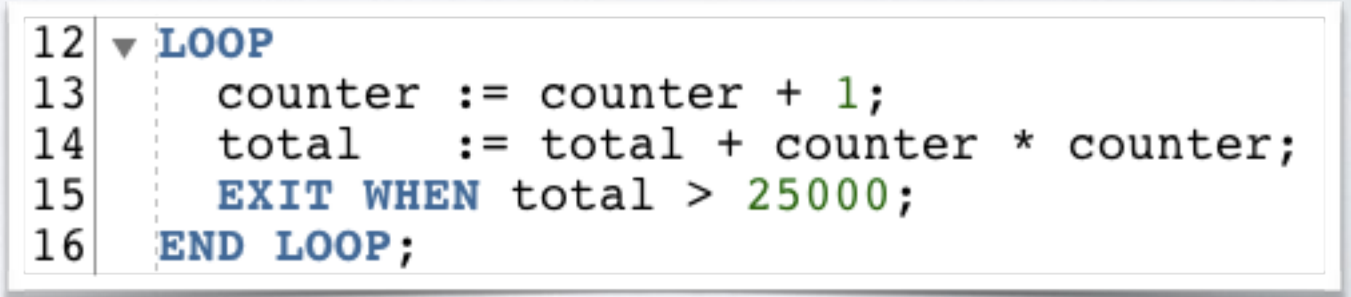
```
LOOP
  --  sequence of statements
END LOOP;
```



```
FOR i IN 1..100 LOOP
  --  Do something here 100 times
END LOOP;
```



```
WHILE sal <= 15000 LOOP
  SELECT salary, manager_id, last_name INTO sal, mgr_id, lname
    FROM employees
    WHERE employee_id = mgr_id;
END LOOP;
```



```
LOOP
  counter := counter + 1;
  total   := total + counter * counter;
  EXIT WHEN total > 25000;
END LOOP;
```

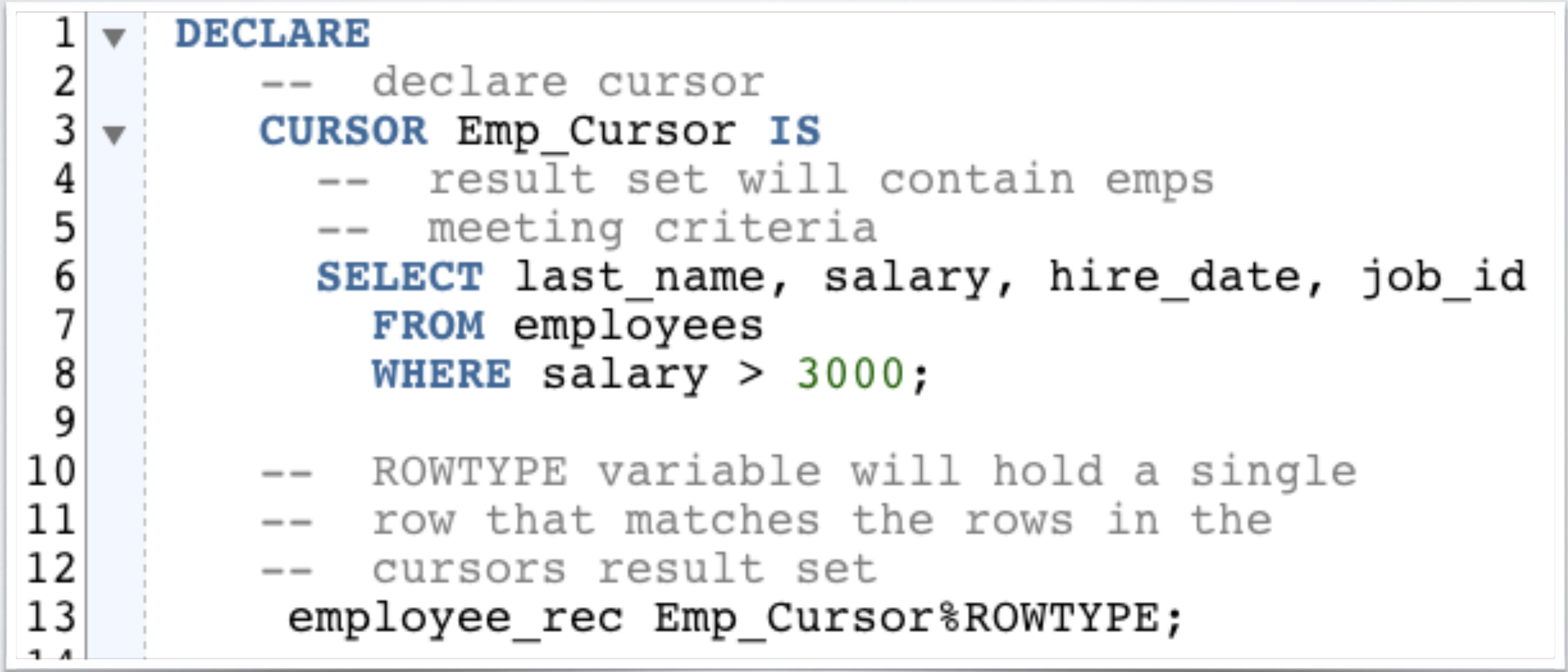
Cursors

- Implicit and Explicit
- Returns a "result set" that you can parse
- Can iterate through result set

Controlling Explicit Cursors

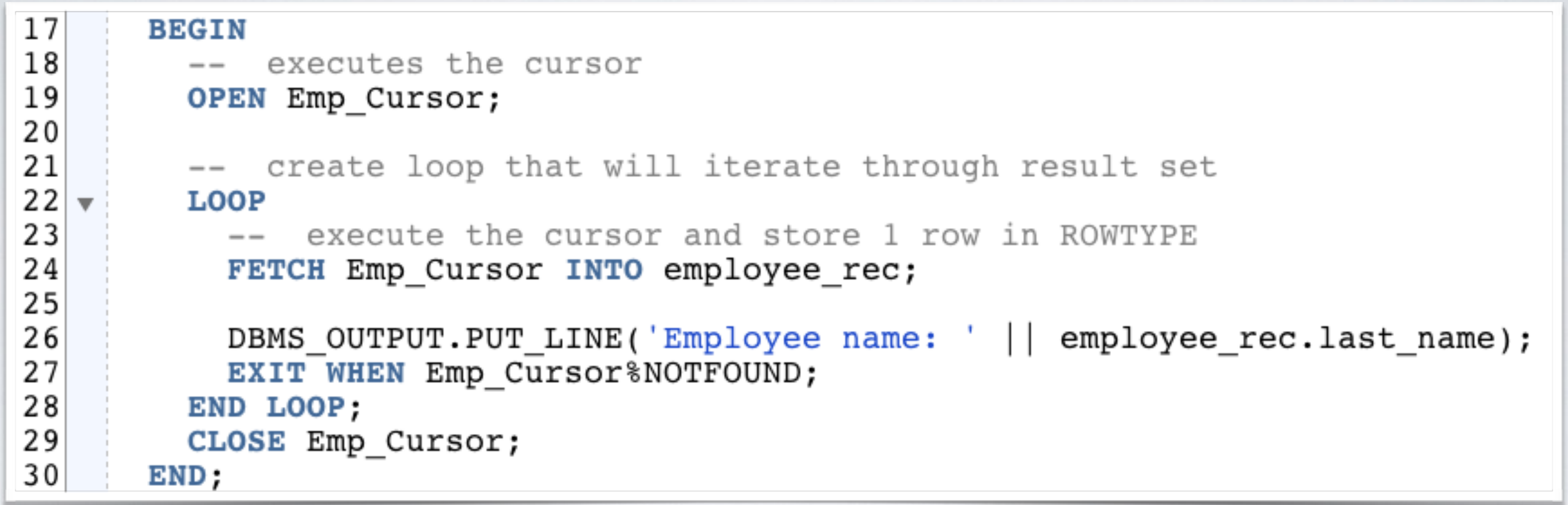
- Declaring - same idea as declaring method in Java
  - Can have optional return type and parameters
- Open - same as calling method in Java
  - Executes the query associate with the cursor
  - Does not put data in result set
- Fetch - similar to iterator in Java
  - Retrieves the current row in result set (one row at a time) and advances cursor to next row
- Close - disables the cursor and makes the result set undefined

Declare Example



```
DECLARE
    --  declare cursor
    CURSOR Emp_Cursor IS
      --  result set will contain emps
      --  meeting criteria
      SELECT last_name, salary, hire_date, job_id
        FROM employees
        WHERE salary > 3000;

    --  ROWTYPE variable will hold a single
    --  row that matches the rows in the
    --  cursors result set
     employee_rec Emp_Cursor%ROWTYPE;
```

Fetch Cursor Example



```
BEGIN
  --  executes the cursor
  OPEN Emp_Cursor;

  --  create loop that will iterate through result set
  LOOP
    --  execute the cursor and store 1 row in ROWTYPE
    FETCH Emp_Cursor INTO employee_rec;

    DBMS_OUTPUT.PUT_LINE('Employee name: ' || employee_rec.last_name);
    EXIT WHEN Emp_Cursor%NOTFOUND;
  END LOOP;
  CLOSE Emp_Cursor;
END;
```

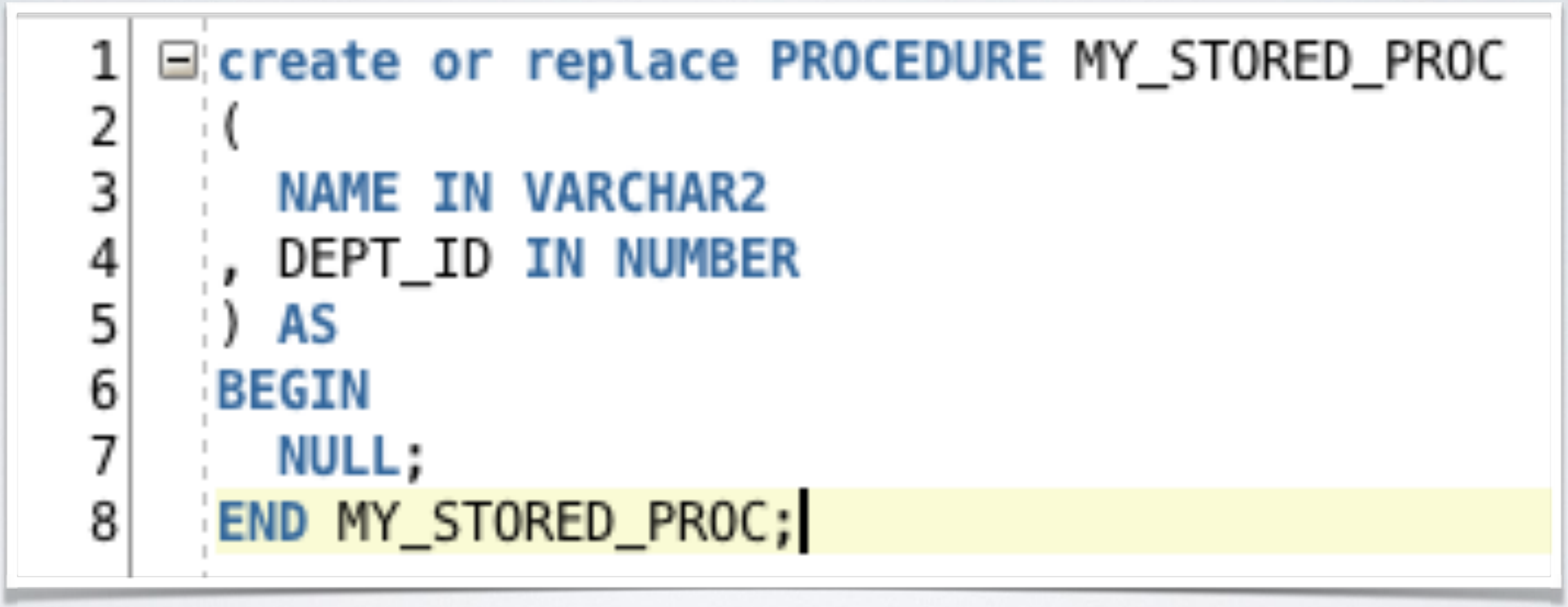
Package

- Bundles subprograms together
- Similar to Java Package
- Specification - Contains subprogram APIs
    - like a .h file in C/C++
- Package Body - Contains the actual subprogram code
- package.subprogram([parameters])

SubPrograms

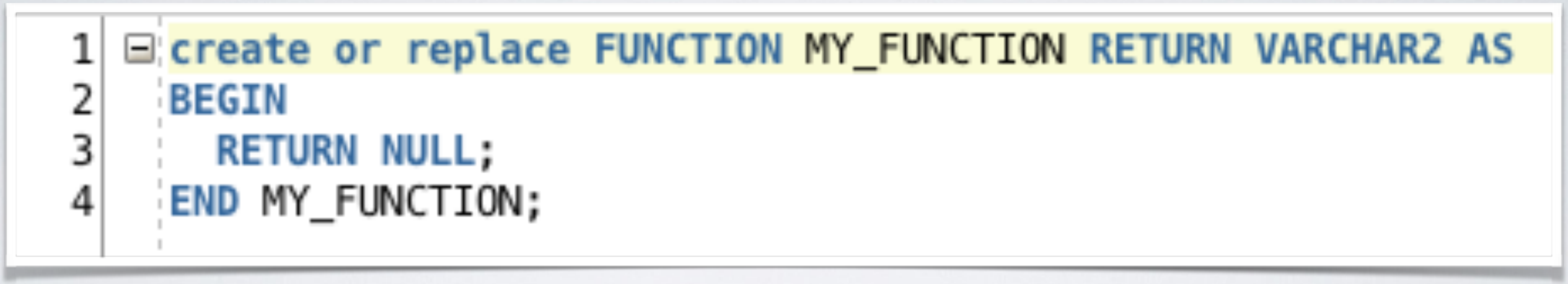
- Anonymous or Named
- Types
  - Procedure - no return value
  - Function - returns value
  - Trigger - executed by DBMS

Stored Procedure



```
create or replace PROCEDURE MY_STORED_PROC
(
  NAME IN VARCHAR2
, DEPT_ID IN NUMBER
) AS
BEGIN
  NULL;
END MY_STORED_PROC;
```

Stored Function

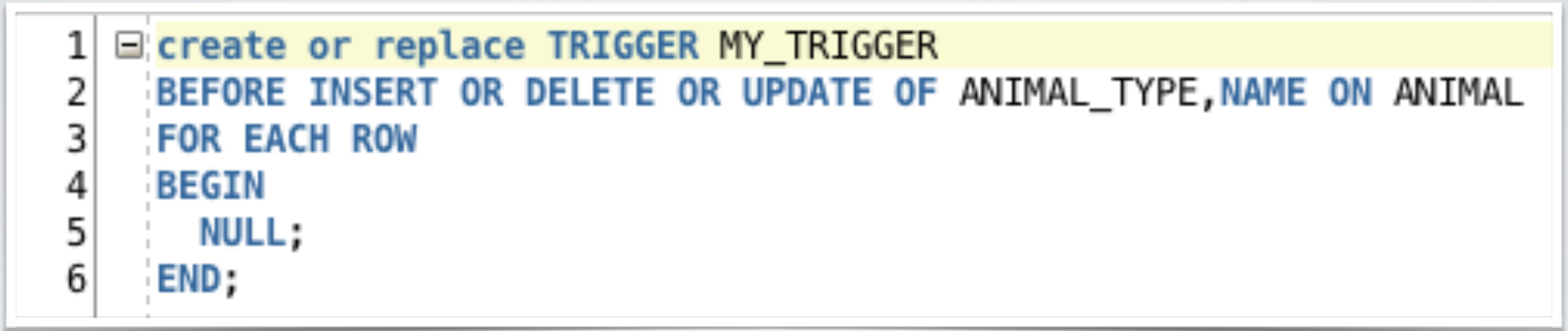


```
create or replace FUNCTION MY_FUNCTION RETURN VARCHAR2 AS
BEGIN
  RETURN NULL;
END MY_FUNCTION;
```

Triggers

- AFTER|BEFORE|INSTEAD OF
- DELETE OR INSERT OR UPDATE
- [FOR EACH ROW]

Trigger

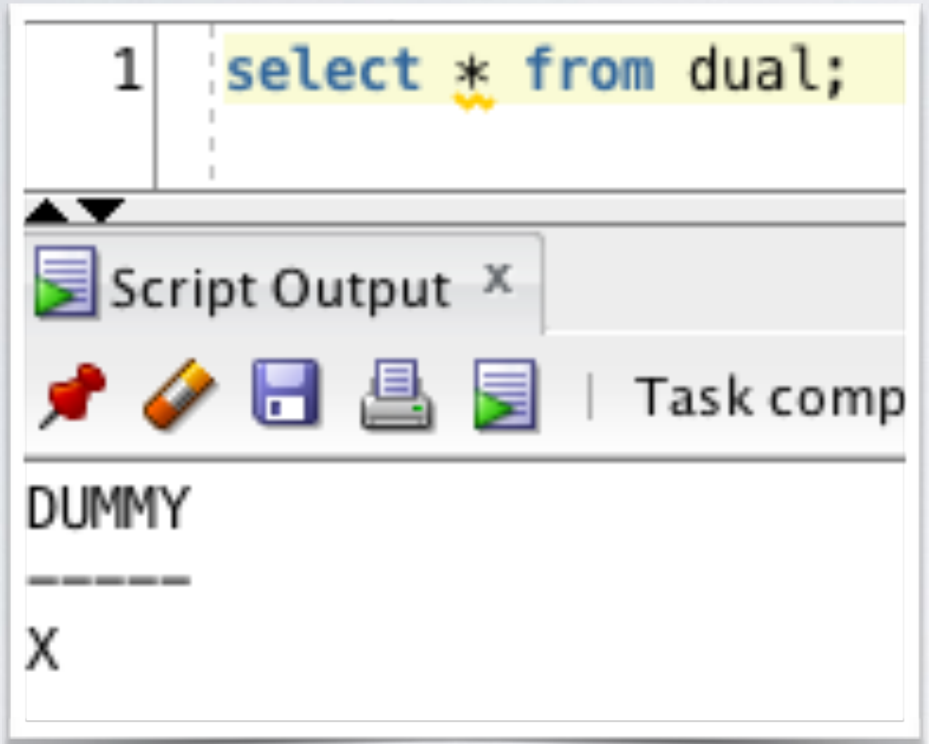


```
create or replace TRIGGER MY_TRIGGER
BEFORE INSERT OR DELETE OR UPDATE OF ANIMAL_TYPE,NAME ON ANIMAL
FOR EACH ROW
BEGIN
  NULL;
END;
```

Dual Table

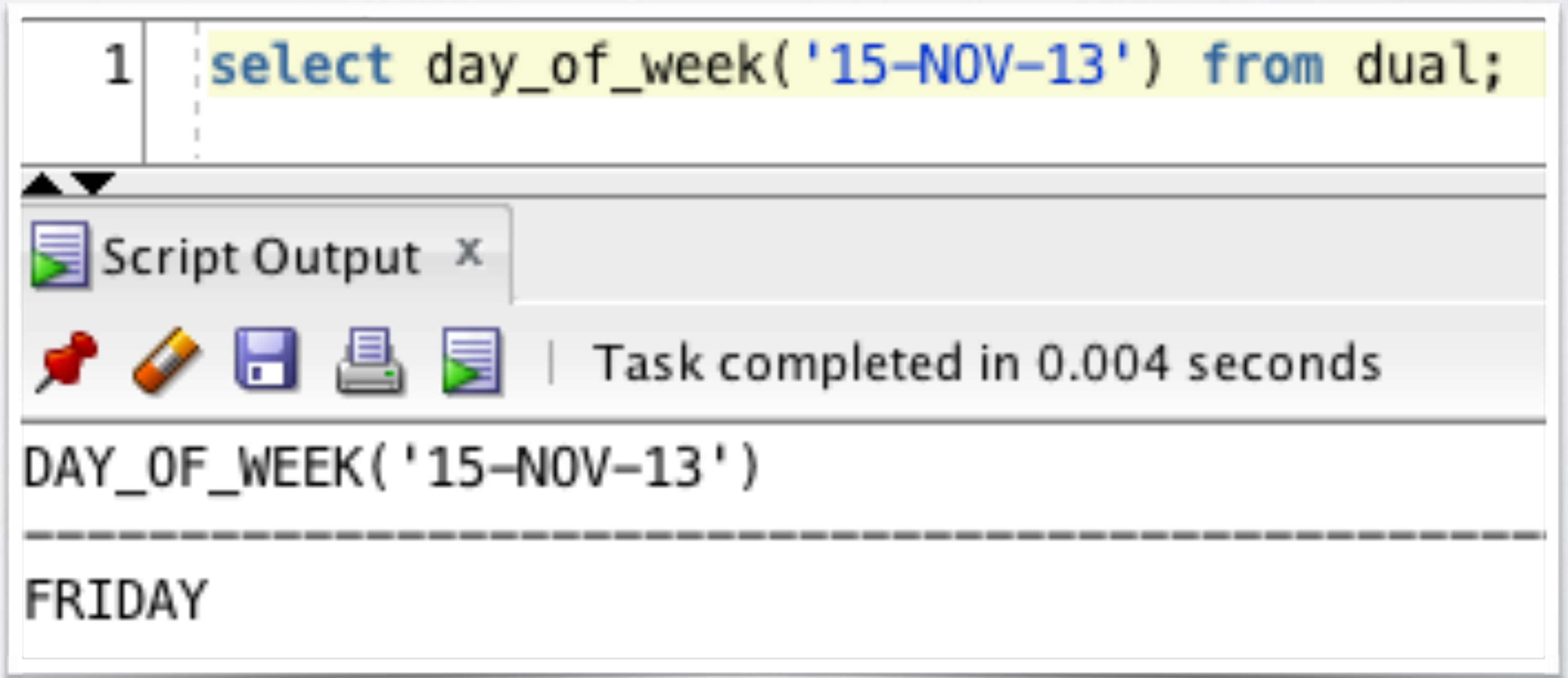
- Dual is a table every user has access to
- Contains
  - 1 column - DUMMY
  - 1 row - value of x

Dual



```
select * from dual;
```

```
DUMMY
-----
X
```

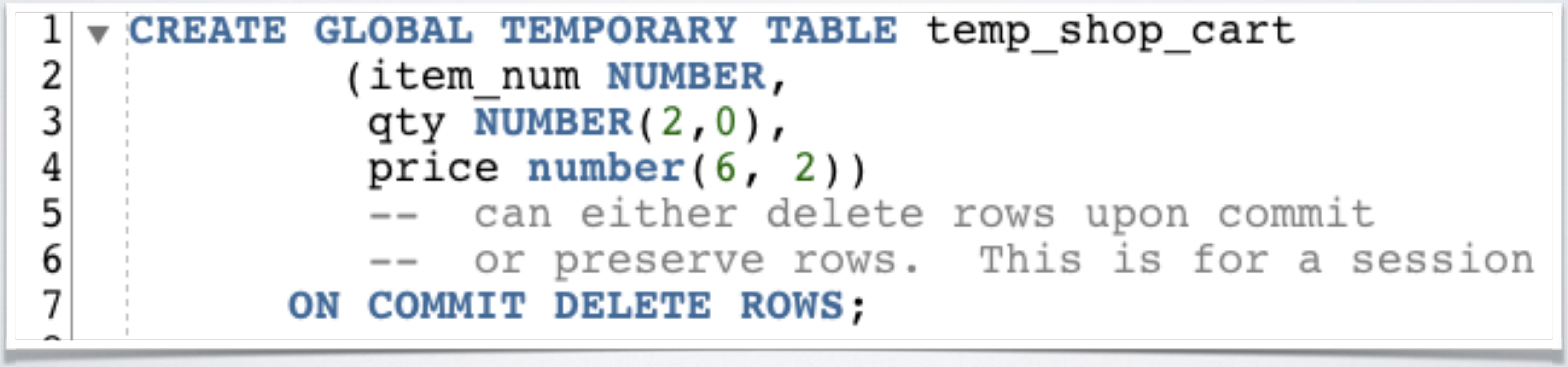


```
select day_of_week('15-NOV-13') from dual;
```

```
DAY_OF_WEEK('15-NOV-13')
----------------------------------------
FRIDAY
```

Temporary Tables

- Used to buffer a result set
- i.e. - Items in shopping cart
    - only need them until order is placed



```
CREATE GLOBAL TEMPORARY TABLE temp_shop_cart
        (item_num NUMBER,
         qty NUMBER(2,0),
         price number(6, 2))
         --  can either delete rows upon commit
         --  or preserve rows.  This is for a session
      ON COMMIT DELETE ROWS;
```