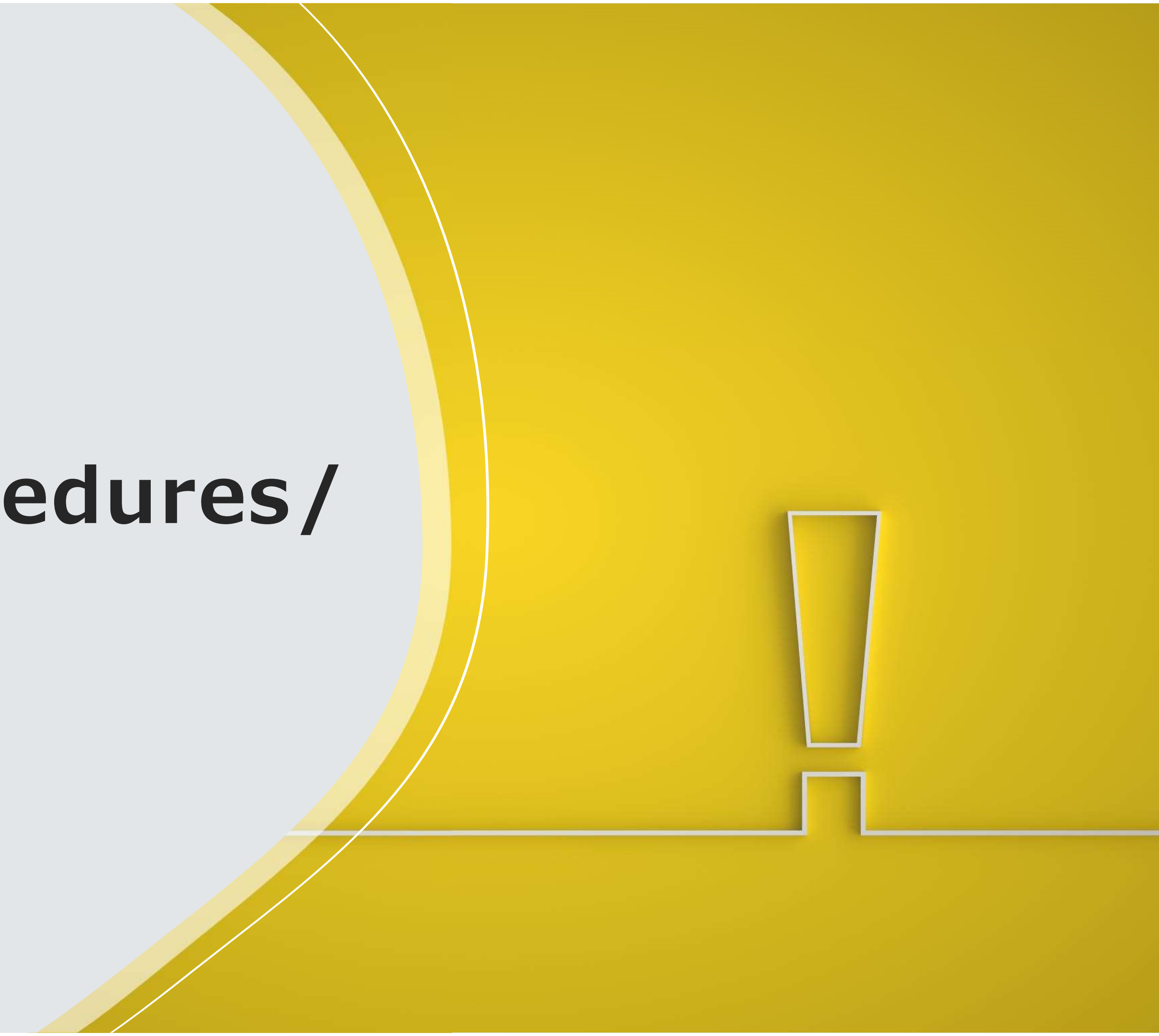
# PL/SQL – Procedures/ Functions



# PL/SQL - Procedures

A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs.
This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

PL/SQL provides two kinds of subprograms −

**Functions** − These subprograms return a single value; mainly used to compute and return a value.

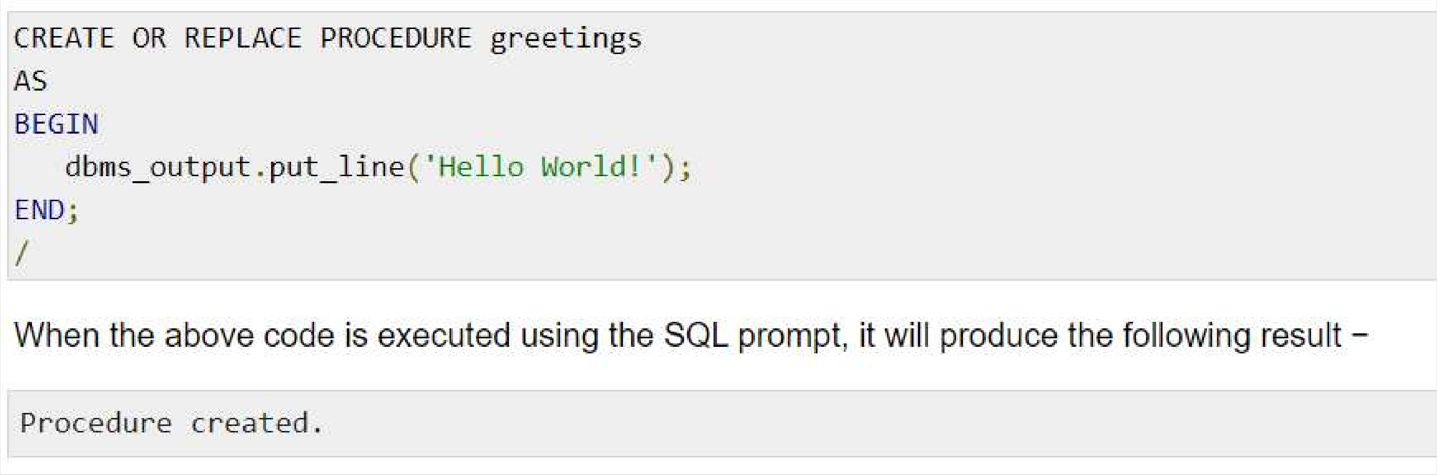
**Procedures** − These subprograms do not return a value directly; mainly used to perform an action

## Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [,
...])] {IS | AS} BEGIN < procedure_body > END
procedure_name;
```

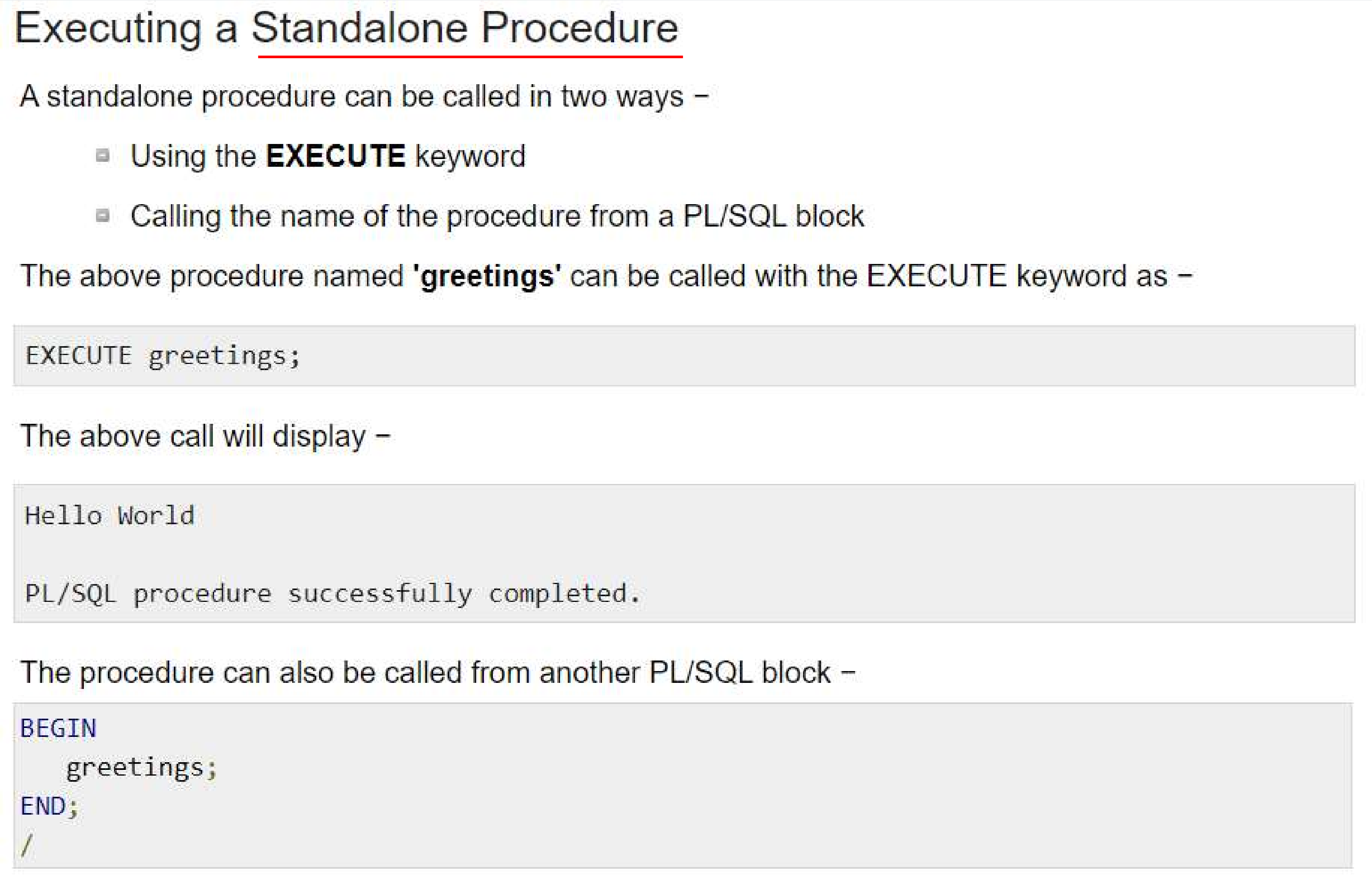
The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.



```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Procedure created.
```



# Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

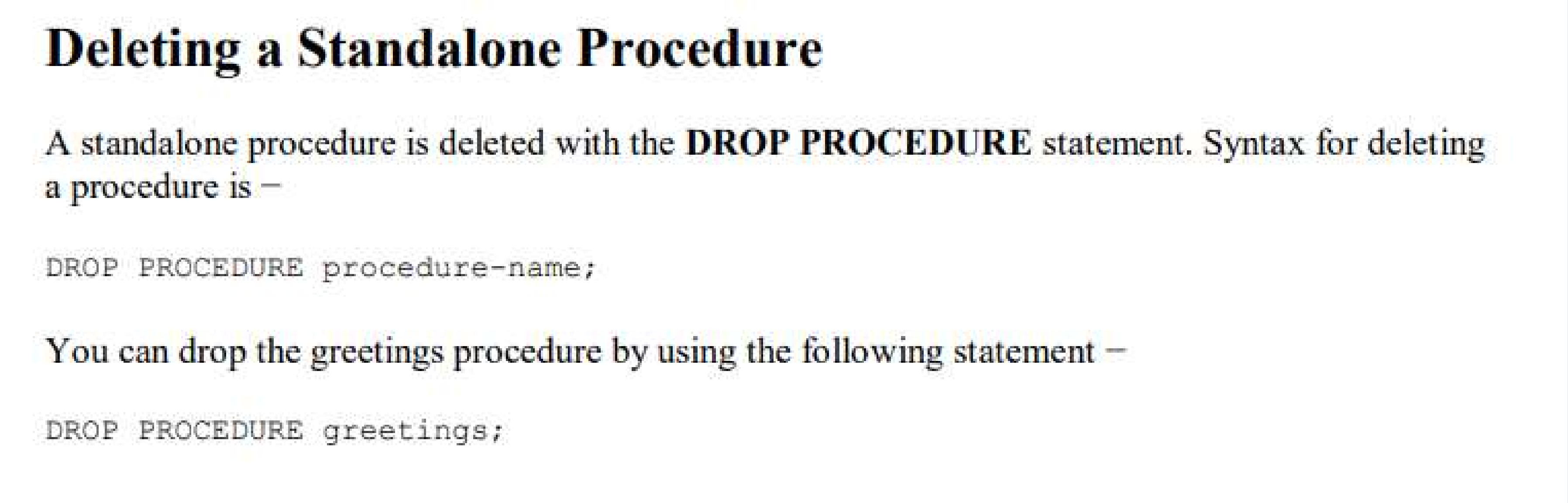
The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
   greetings;
END;
/
```



## Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

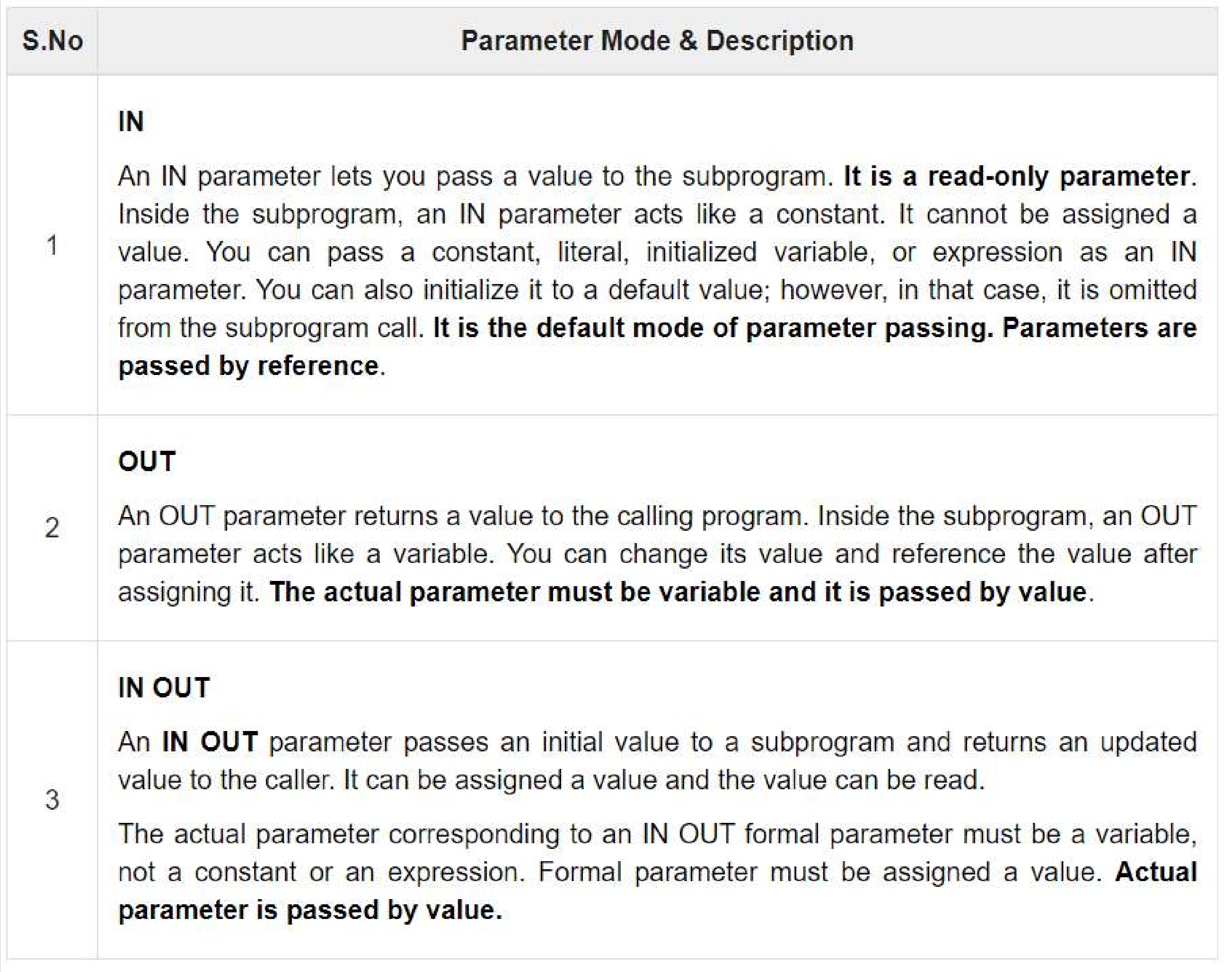
```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement −

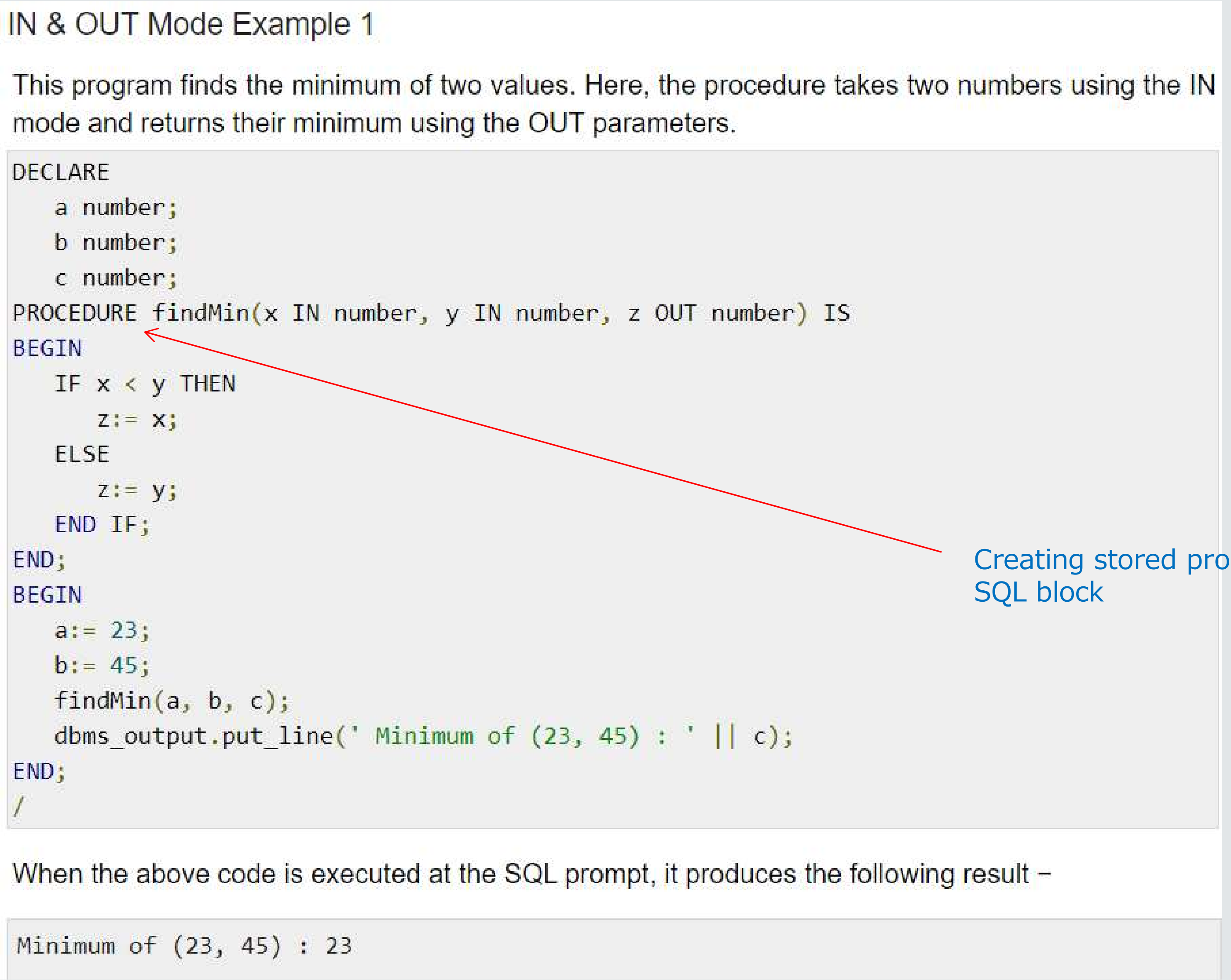
```
DROP PROCEDURE greetings;
```

# Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −



| S.No | Parameter Mode & Description |
|---|---|
| 1 | **IN**<br>An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**. |
| 2 | **OUT**<br>An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**. |
| 3 | **IN OUT**<br>An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.<br>The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.** |



## IN & OUT Mode Example 1

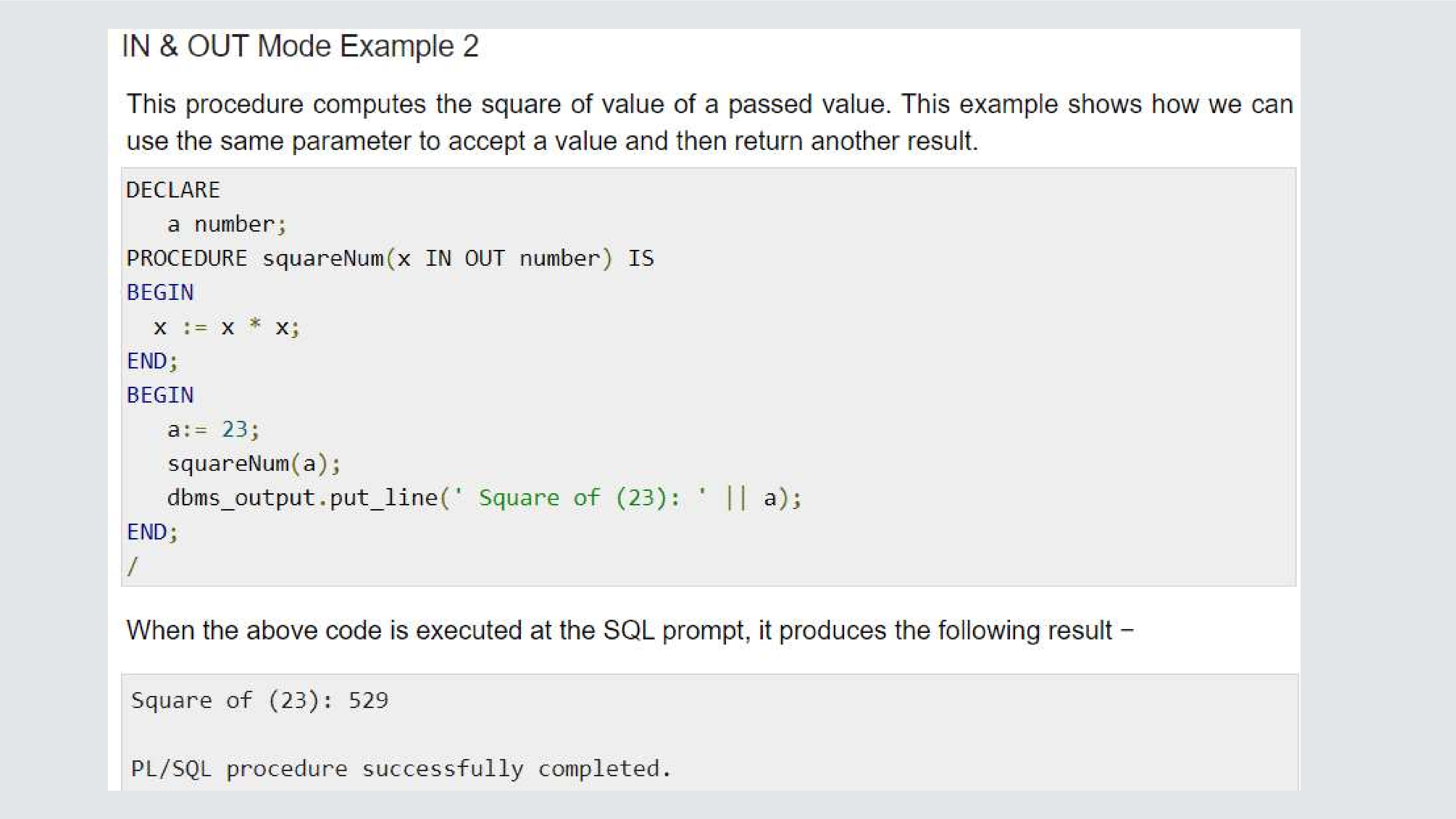
This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
   a number;
   b number;
   c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;
BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Minimum of (23, 45) : 23
```

Creating stored procedure in PL SQL block



## IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

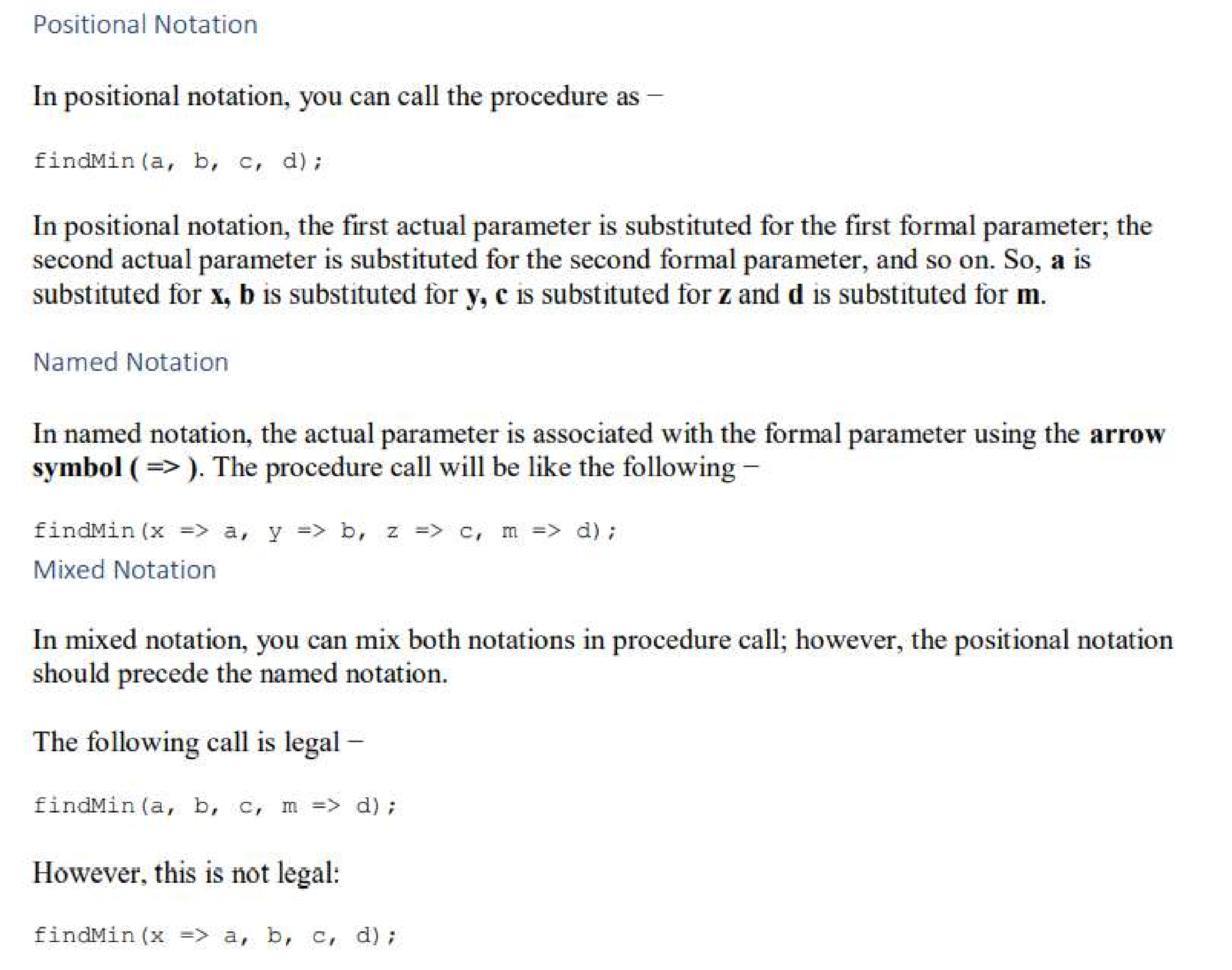
```
Square of (23): 529

PL/SQL procedure successfully completed.
```

# Methods for Passing Parameters

Actual parameters can be passed in three ways –

- Positional notation
- Named notation
- Mixed notation



### Positional Notation

In positional notation, you can call the procedure as –

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

### Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol ( => )**. The procedure call will be like the following –

```
findMin(x => a, y => b, z => c, m => d);
```

### Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal –

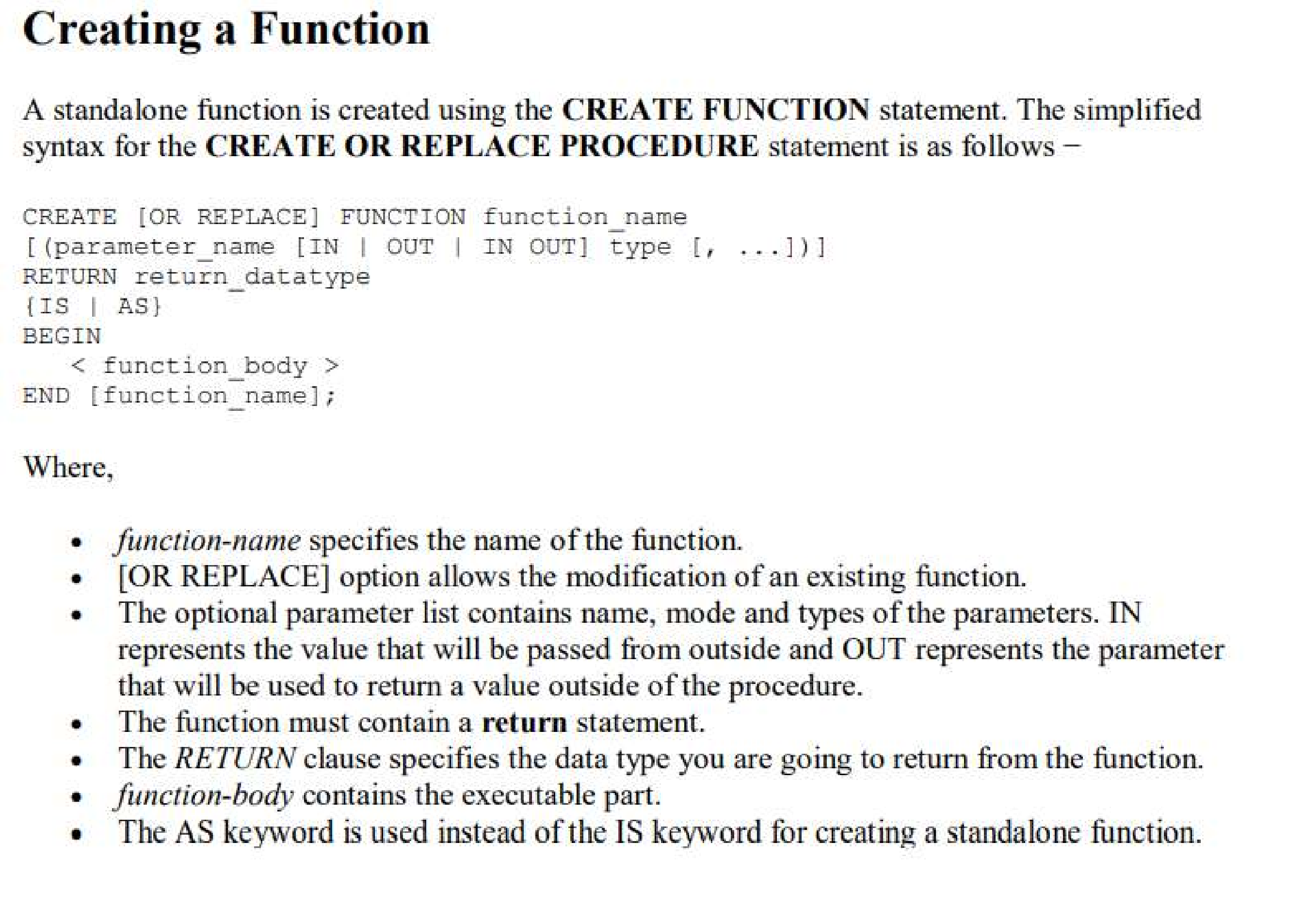
```
findMin(a, b, c, m => d);
```

However, this is not legal:

```
findMin(x => a, b, c, d);
```

# PL/SQL – Functions

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous sections are true for functions too.



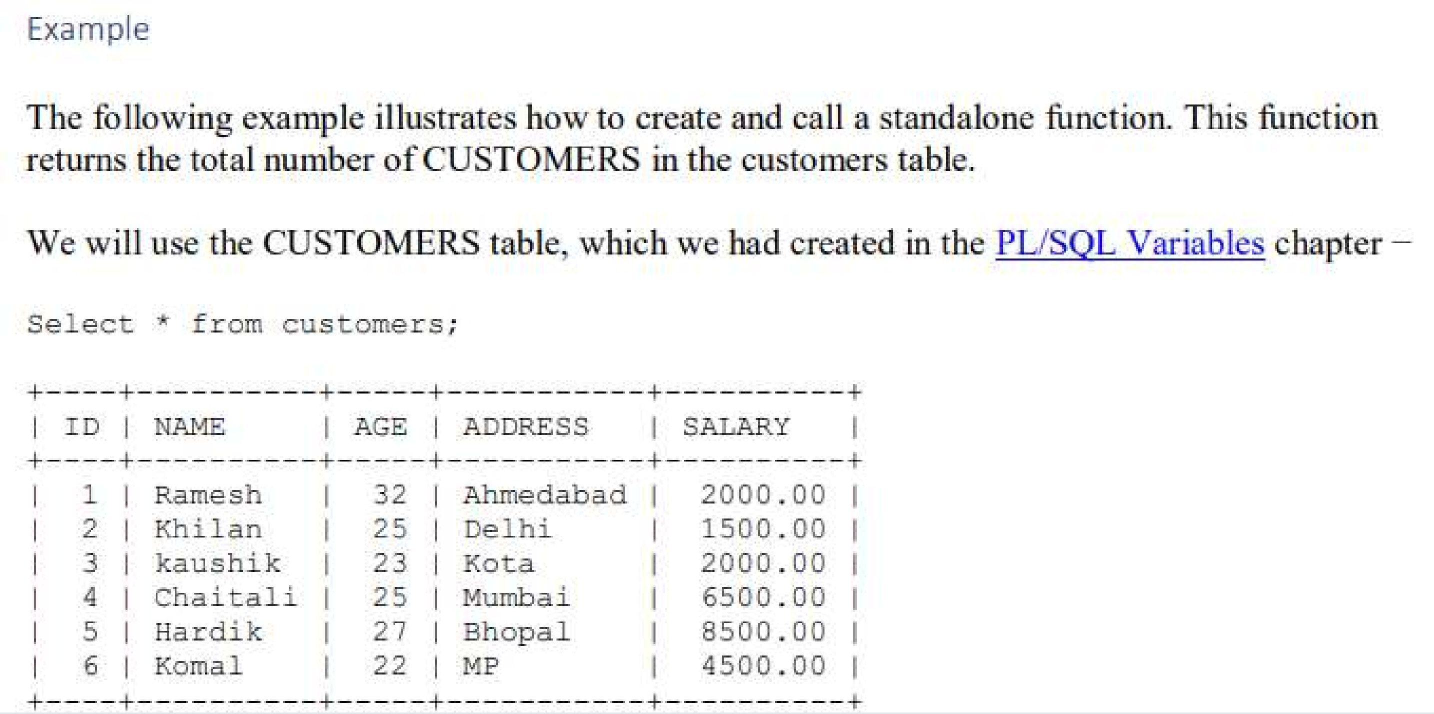
## Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.



Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter –

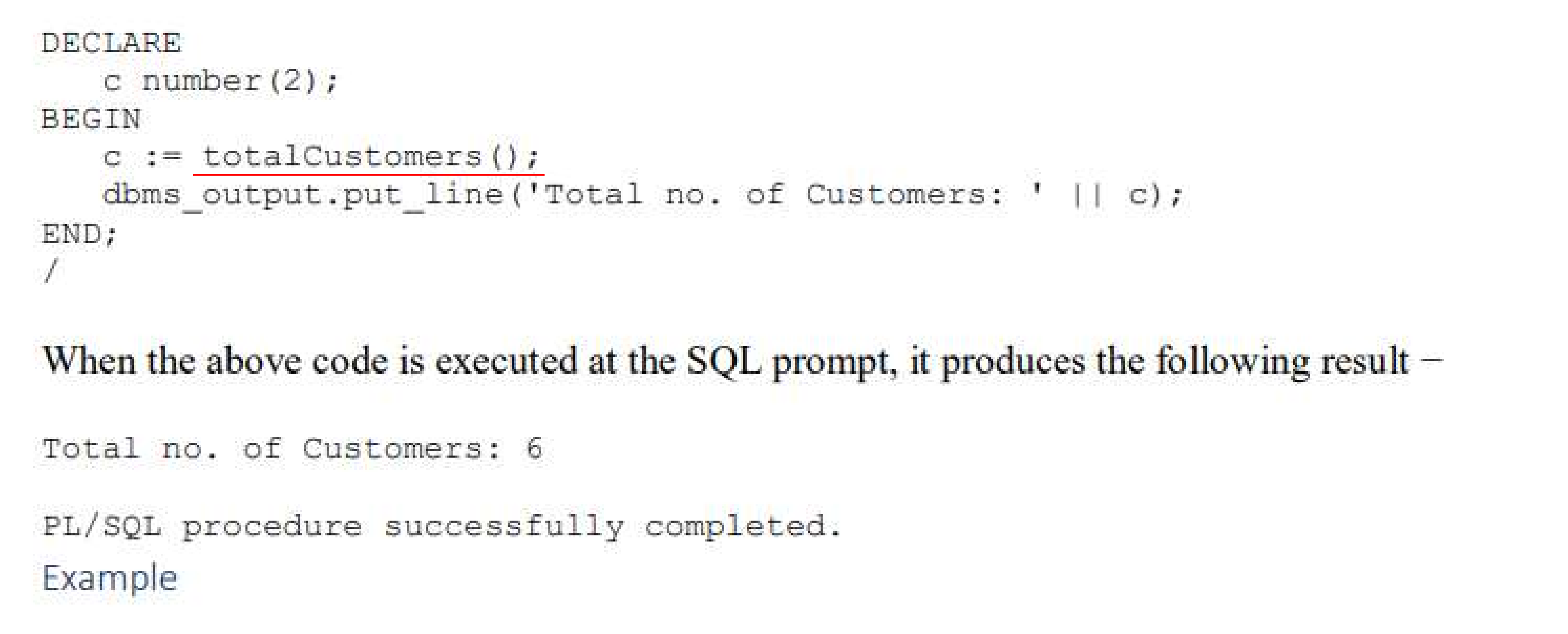
```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
total number(2) := 0;
BEGIN
SELECT count(*) into total FROM customers;
RETURN total;
END;
```

/ When the above code is executed using the SQL prompt, it will produce the following result – Function created

# Calling a function



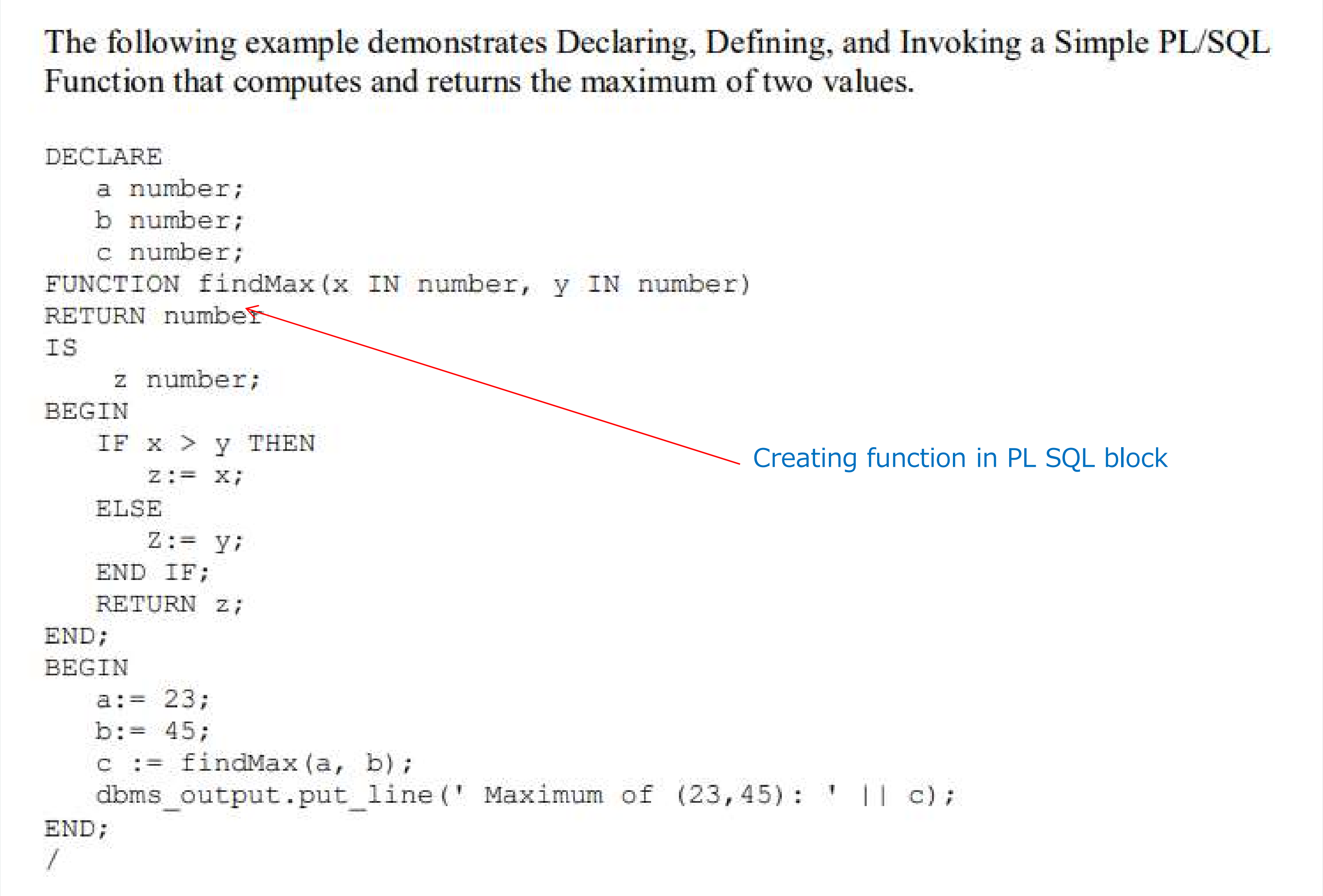
```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example



The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

Creating function in PL SQL block

# Procedure Vs. Function: Key Differences

| Procedure | Function |
|---|---|
| •Used mainly to a execute certain process | •Used mainly to perform some calculation |
| •Cannot call in SELECT statement | •A Function that contains no DML statements can be called in SELECT statement |
| •Use OUT parameter to return the value | •Use RETURN to return the value |
| •It is not mandatory to return the value | •It is mandatory to return the value |
| •RETURN will simply exit the control from subprogram. | •RETURN will exit the control from subprogram and also returns the value |
| •Return datatype will not be specified at the time of creation | •Return datatype is mandatory at the time of creation |

# PL/SQL - Records

A record is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

PL/SQL can handle the following types of records

- Table-based
- Cursor-based records
- User-defined records

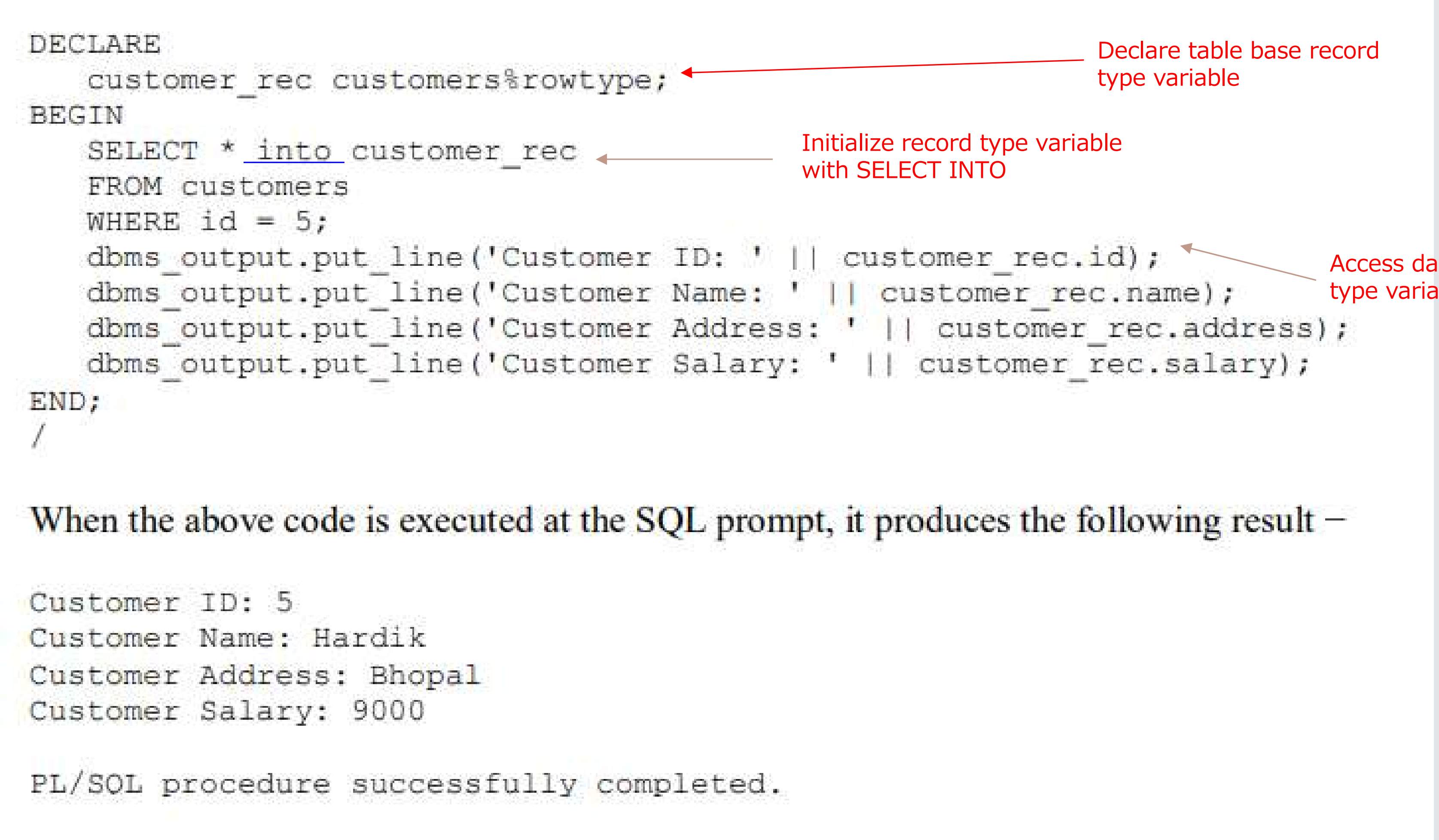
## What are Table Based Record Datatype Variables?

As the name suggests Table Based Record Datatype Variables are variable of record datatype created over a table of your database.

## Table-Based Records

The **%ROWTYPE** attribute enables a programmer to create **table-based** records.

The following example illustrates the concept of table-based records. We will be using the CUSTOMERS table we had created and used in the previous chapters −



```
DECLARE
   customer_rec customers%rowtype;
BEGIN
   SELECT * into customer_rec
   FROM customers
   WHERE id = 5;
   dbms_output.put_line('Customer ID: ' || customer_rec.id);
   dbms_output.put_line('Customer Name: ' || customer_rec.name);
   dbms_output.put_line('Customer Address: ' || customer_rec.address);
   dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000

PL/SQL procedure successfully completed.
```

Declare table base record
type variable

Initialize record type variable
with SELECT INTO

Access date from the record
type variable

# Cursor Based Records in PL SQL

Cursors in PL/SQL. Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the **number of rows processed**, etc.

A cursor is a pointer to this context area.
PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time.

There are two types of cursors −

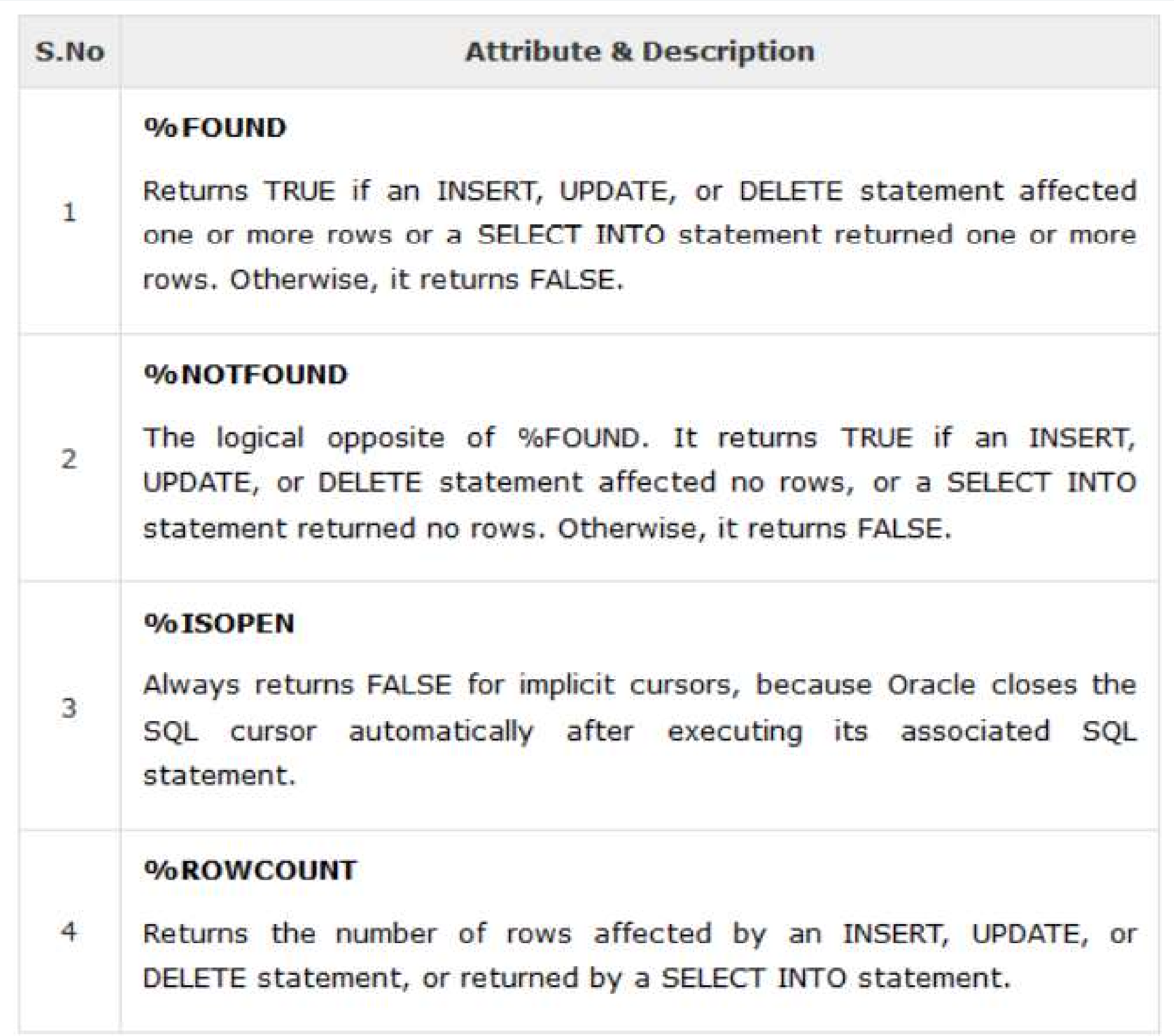
- **Implicit cursors**
- **Explicit cursors**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.

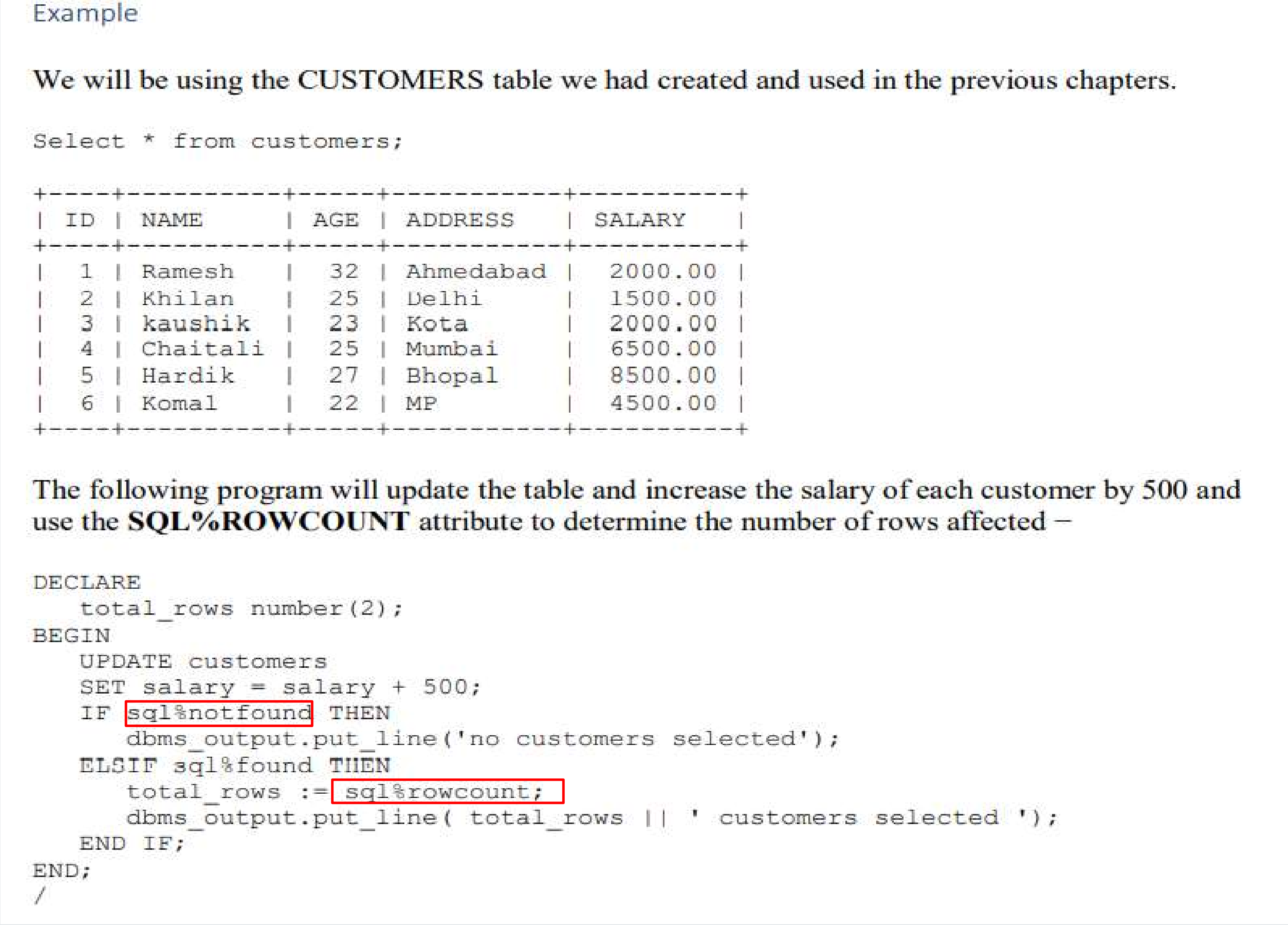
Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected

In PL/SQL, you can refer to the most recent **implicit cursor** as the SQL cursor,

which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND,** and **%ROWCOUNT**



| S.No | Attribute & Description |
|---|---|
| 1 | **%FOUND**<br>Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| 2 | **%NOTFOUND**<br>The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| 3 | **%ISOPEN**<br>Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| 4 | **%ROWCOUNT**<br>Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |



Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

## Explicit Cursors

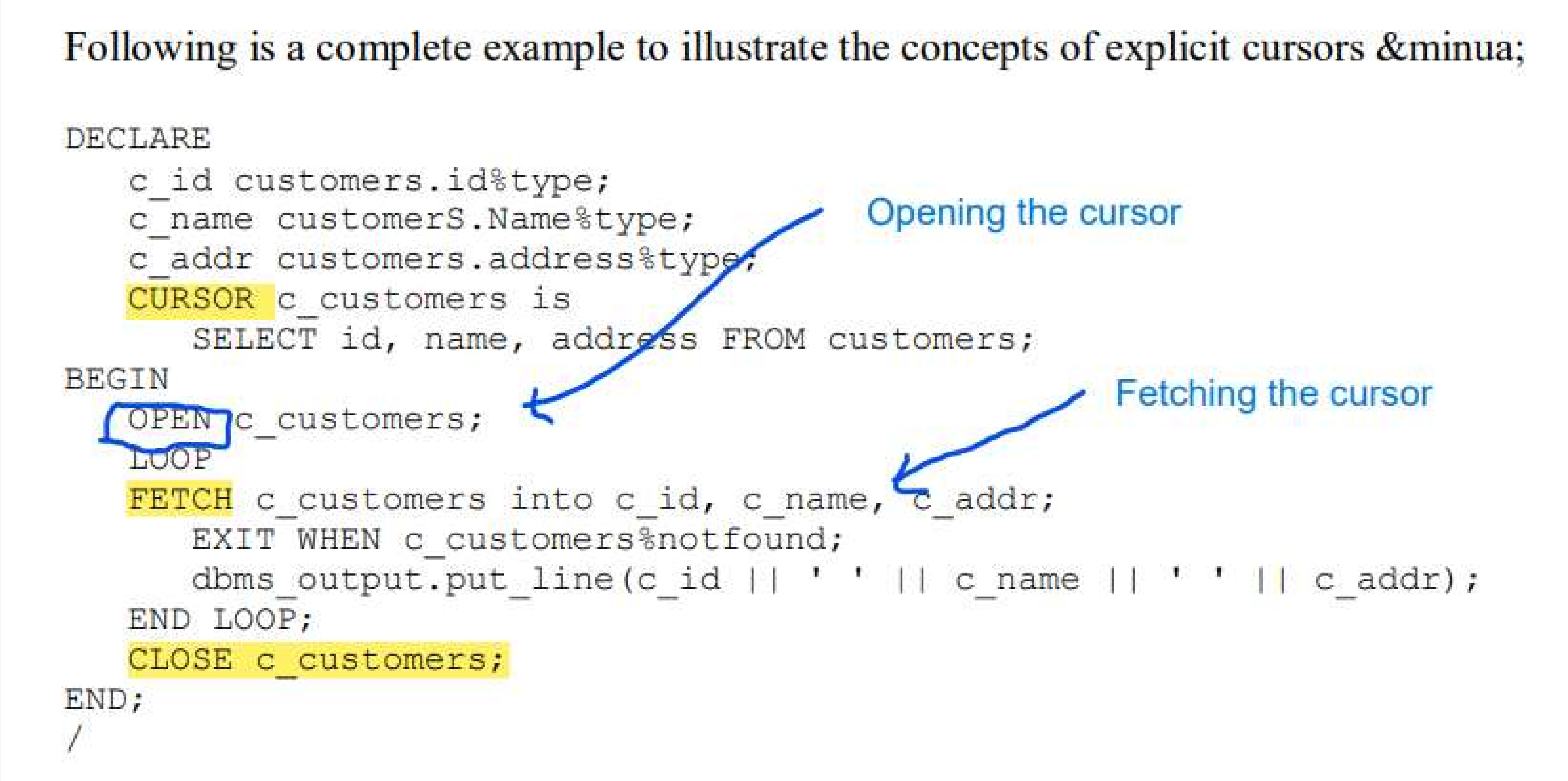
Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Declaring the Cursor Declaring the cursor defines the cursor with a name and the associated SELECT statement.

For example – CURSOR c_customers IS SELECT id, name, address FROM customers;

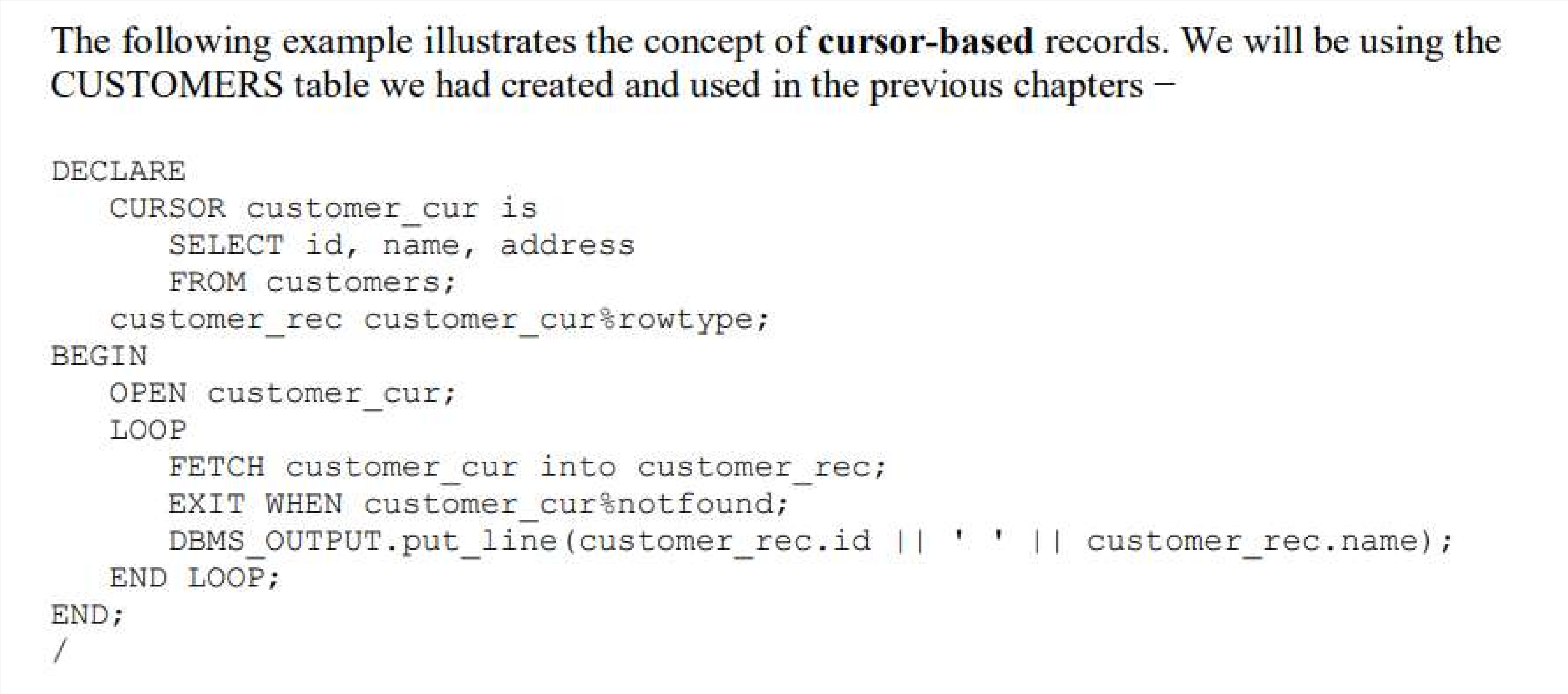


Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customerS.Name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

Opening the cursor

Fetching the cursor



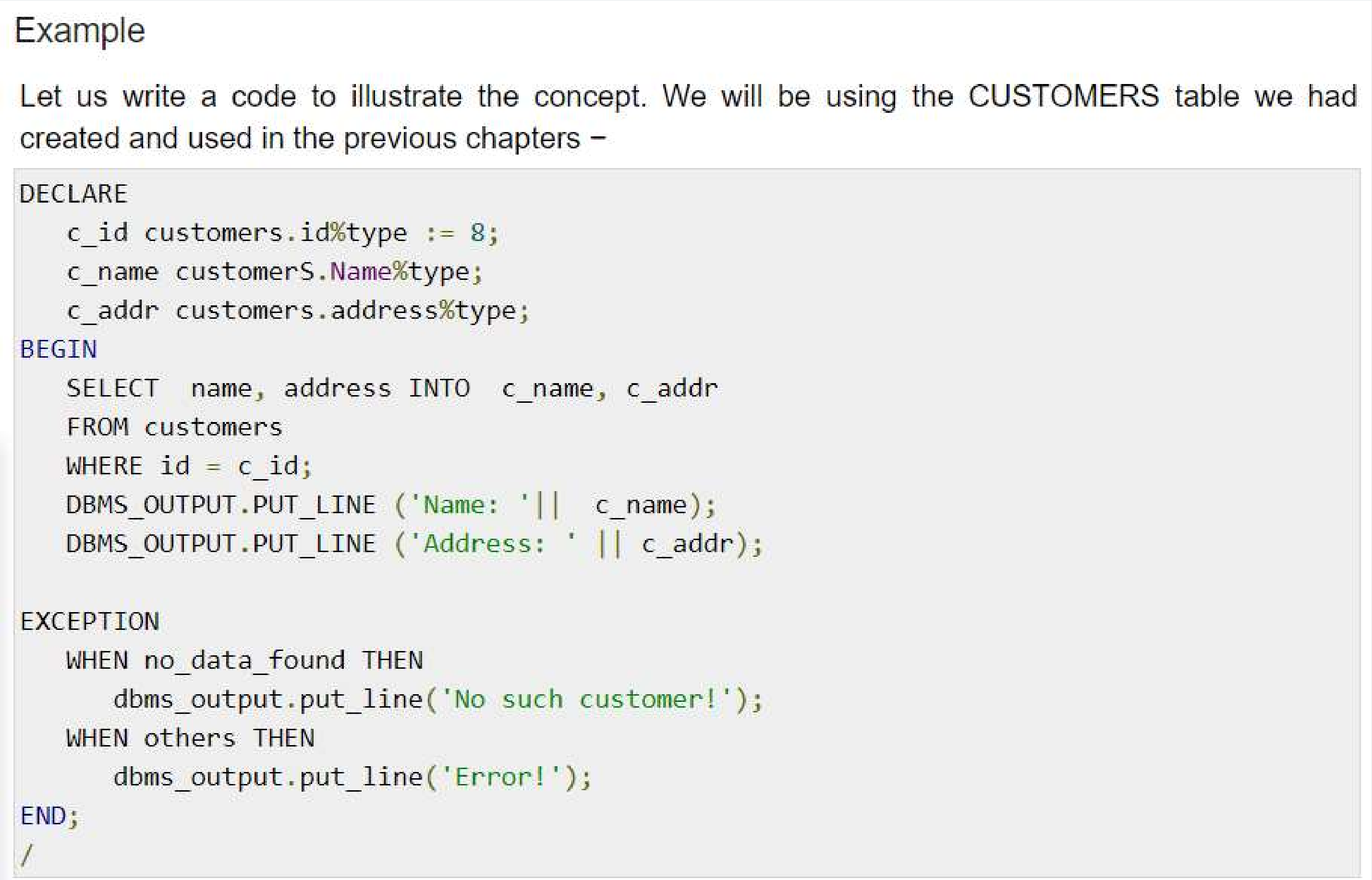
The following example illustrates the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters −

```
DECLARE
   CURSOR customer_cur is
      SELECT id, name, address
      FROM customers;
   customer_rec customer_cur%rowtype;
BEGIN
   OPEN customer_cur;
   LOOP
      FETCH customer_cur into customer_rec;
      EXIT WHEN customer_cur%notfound;
      DBMS_OUTPUT.put_line(customer_rec.id || ' ' || customer_rec.name);
   END LOOP;
END;
/
```

# PL/SQL - Exceptions

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition. There are two types of exceptions –

- **System-defined exceptions**
- **User-defined exception**



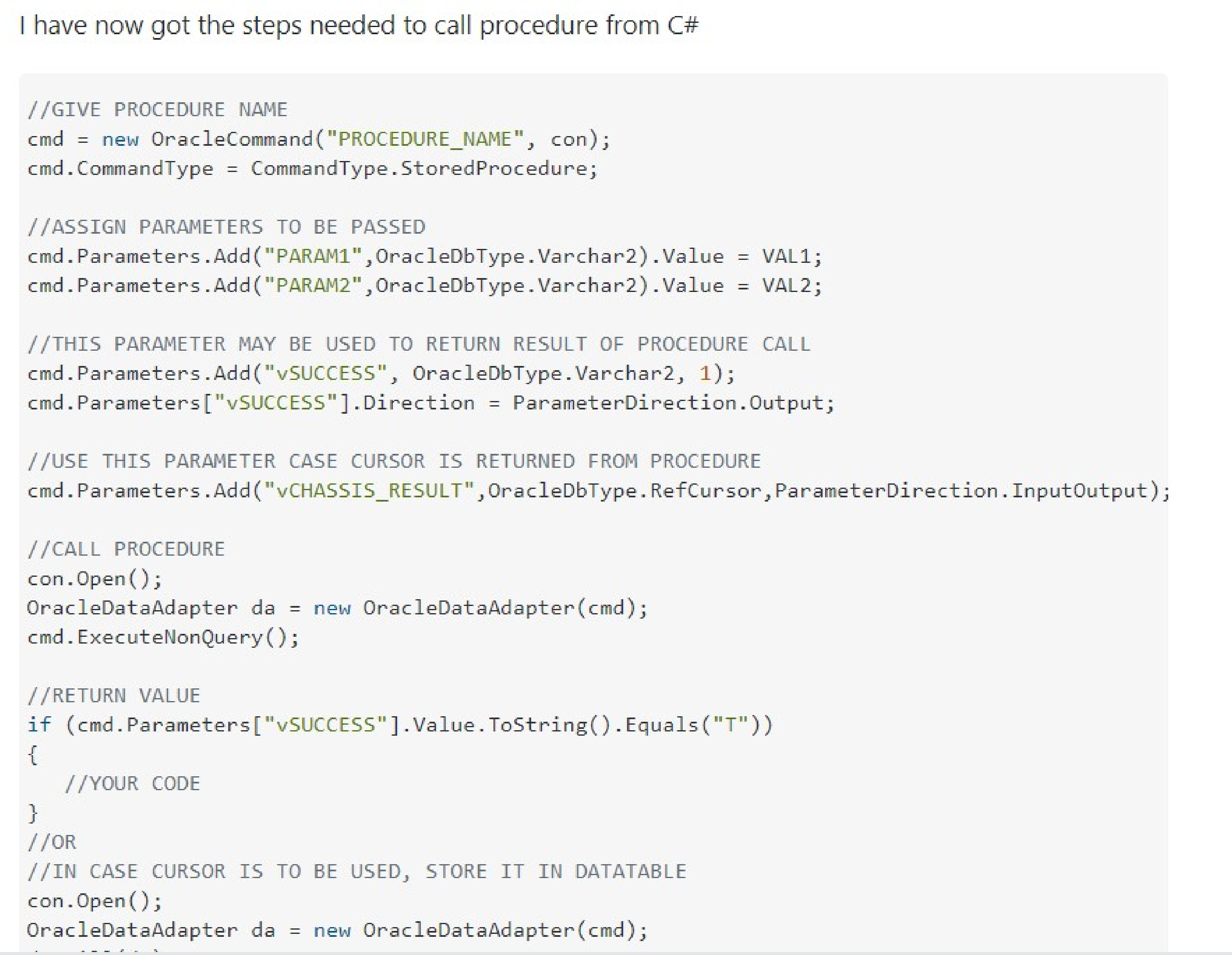
## Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters –

```
DECLARE
   c_id customers.id%type := 8;
   c_name customerS.Name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

# Calling PL SQL function from C#.Net code



I have now got the steps needed to call procedure from C#

```
//GIVE PROCEDURE NAME
cmd = new OracleCommand("PROCEDURE_NAME", con);
cmd.CommandType = CommandType.StoredProcedure;

//ASSIGN PARAMETERS TO BE PASSED
cmd.Parameters.Add("PARAM1",OracleDbType.Varchar2).Value = VAL1;
cmd.Parameters.Add("PARAM2",OracleDbType.Varchar2).Value = VAL2;

//THIS PARAMETER MAY BE USED TO RETURN RESULT OF PROCEDURE CALL
cmd.Parameters.Add("vSUCCESS", OracleDbType.Varchar2, 1);
cmd.Parameters["vSUCCESS"].Direction = ParameterDirection.Output;

//USE THIS PARAMETER CASE CURSOR IS RETURNED FROM PROCEDURE
cmd.Parameters.Add("vCHASSIS_RESULT",OracleDbType.RefCursor,ParameterDirection.InputOutput);

//CALL PROCEDURE
con.Open();
OracleDataAdapter da = new OracleDataAdapter(cmd);
cmd.ExecuteNonQuery();

//RETURN VALUE
if (cmd.Parameters["vSUCCESS"].Value.ToString().Equals("T"))
{
   //YOUR CODE
}
//OR
//IN CASE CURSOR IS TO BE USED, STORE IT IN DATATABLE
con.Open();
OracleDataAdapter da = new OracleDataAdapter(cmd);
```