Exercise 3: Integrated Languages and PL / SQL

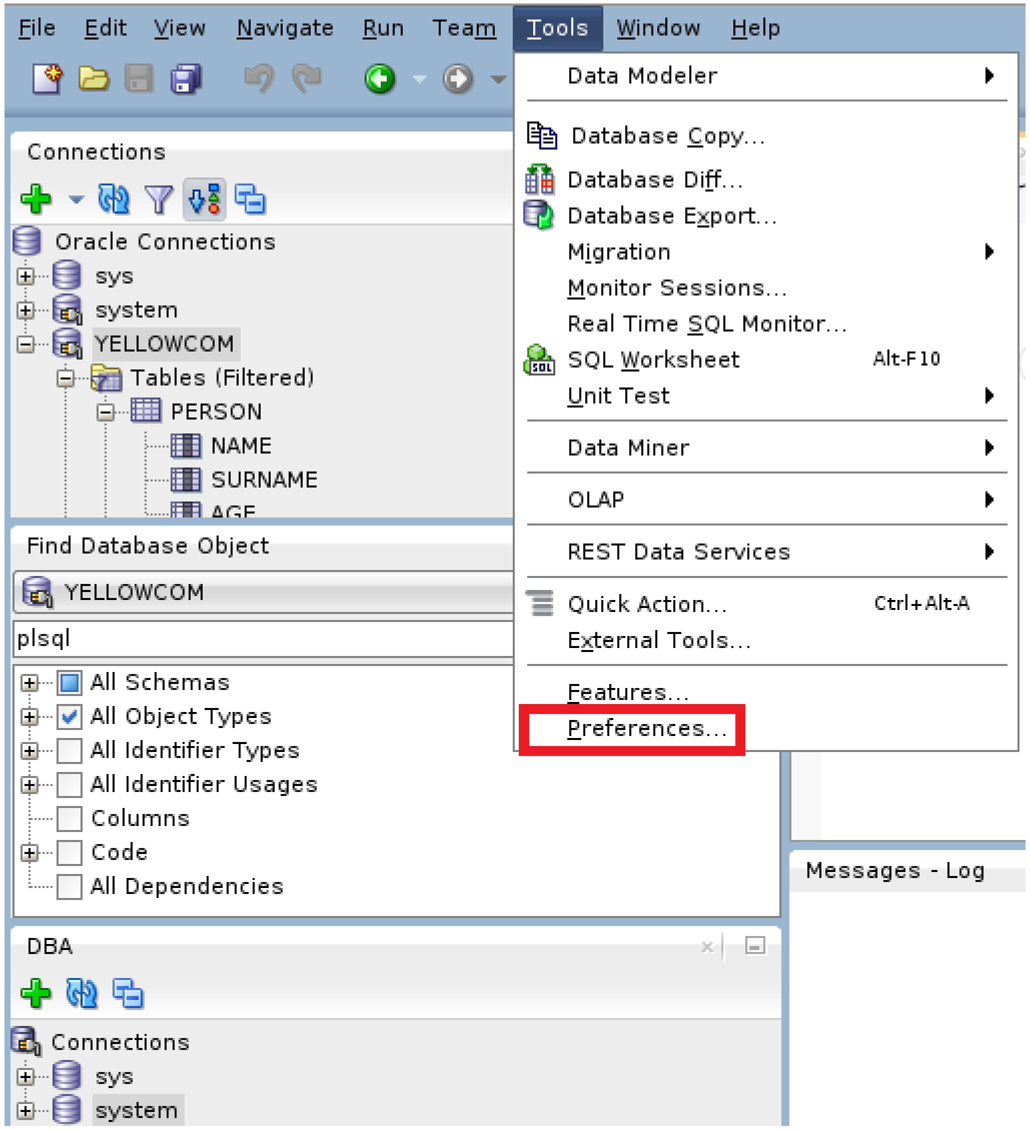
Integrated Languages

PL / SQL: It is a structured query language (Oracle SQL), extended with a procedural language (PL, Procedural Language) .

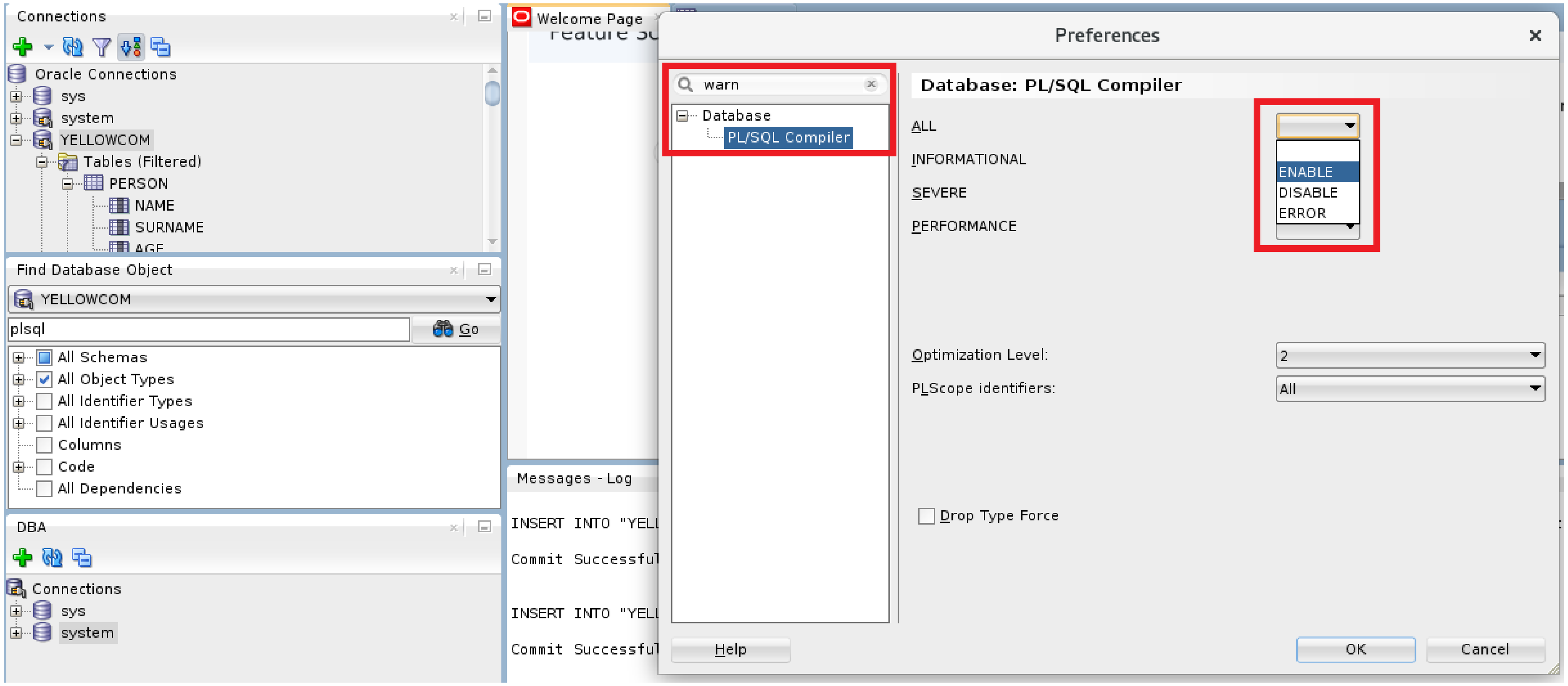
PL / SQL can be used to encode the business rules by creating stored procedures and packages to:

- add a programming logic execution of SQL commands;
- enable database events when necessary (see the triggers of active databases).

Enable PL / SQL



Enable PL / SQL



Preferences

warn

Database
PL/SQL Compiler

Database: PL/SQL Compiler

ALL
INFORMATIONAL
SEVERE
PERFORMANCE

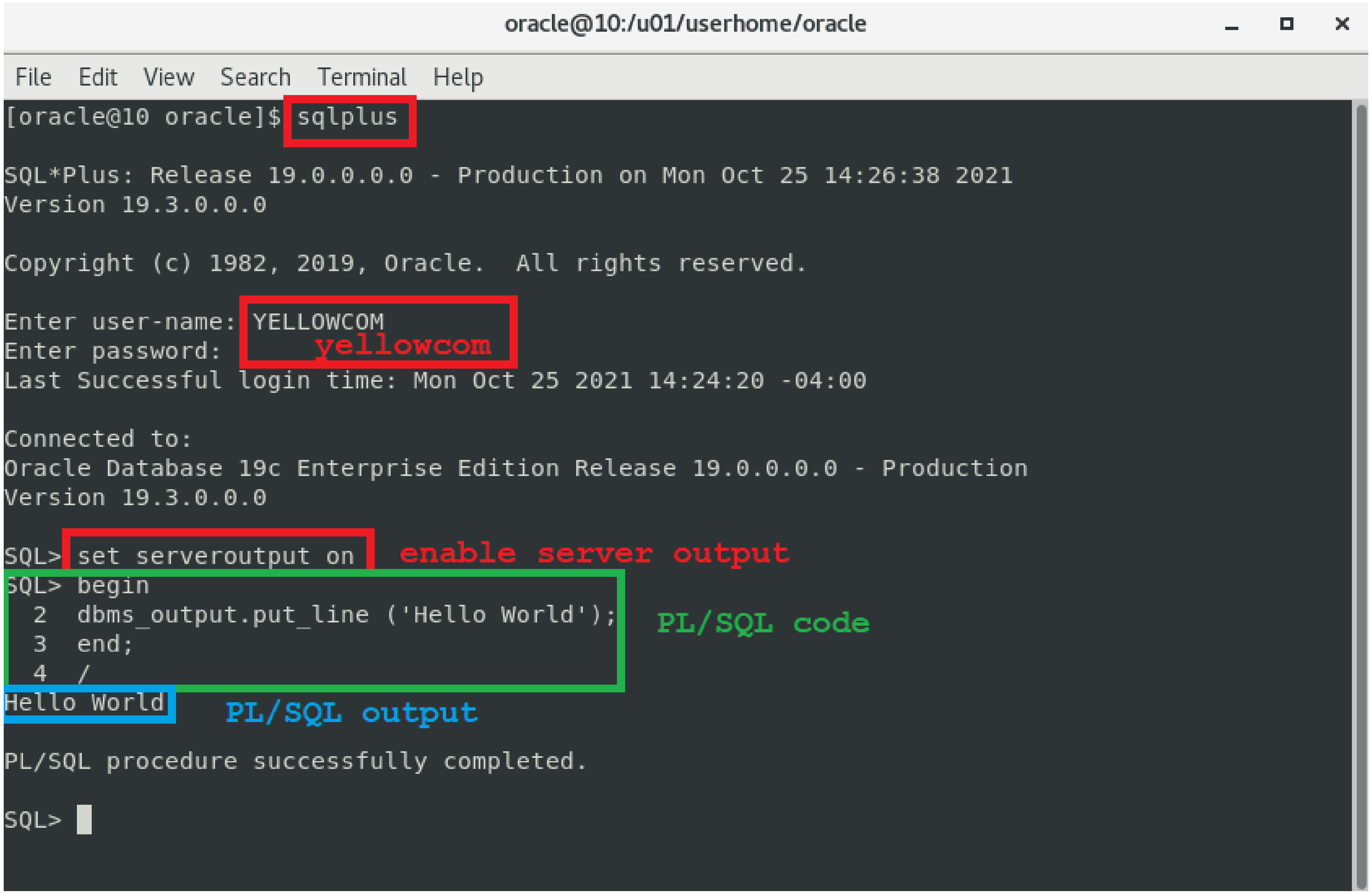
ENABLE
DISABLE
ERROR

Optimization Level: 2
PLScope identifiers: All

Drop Type Force

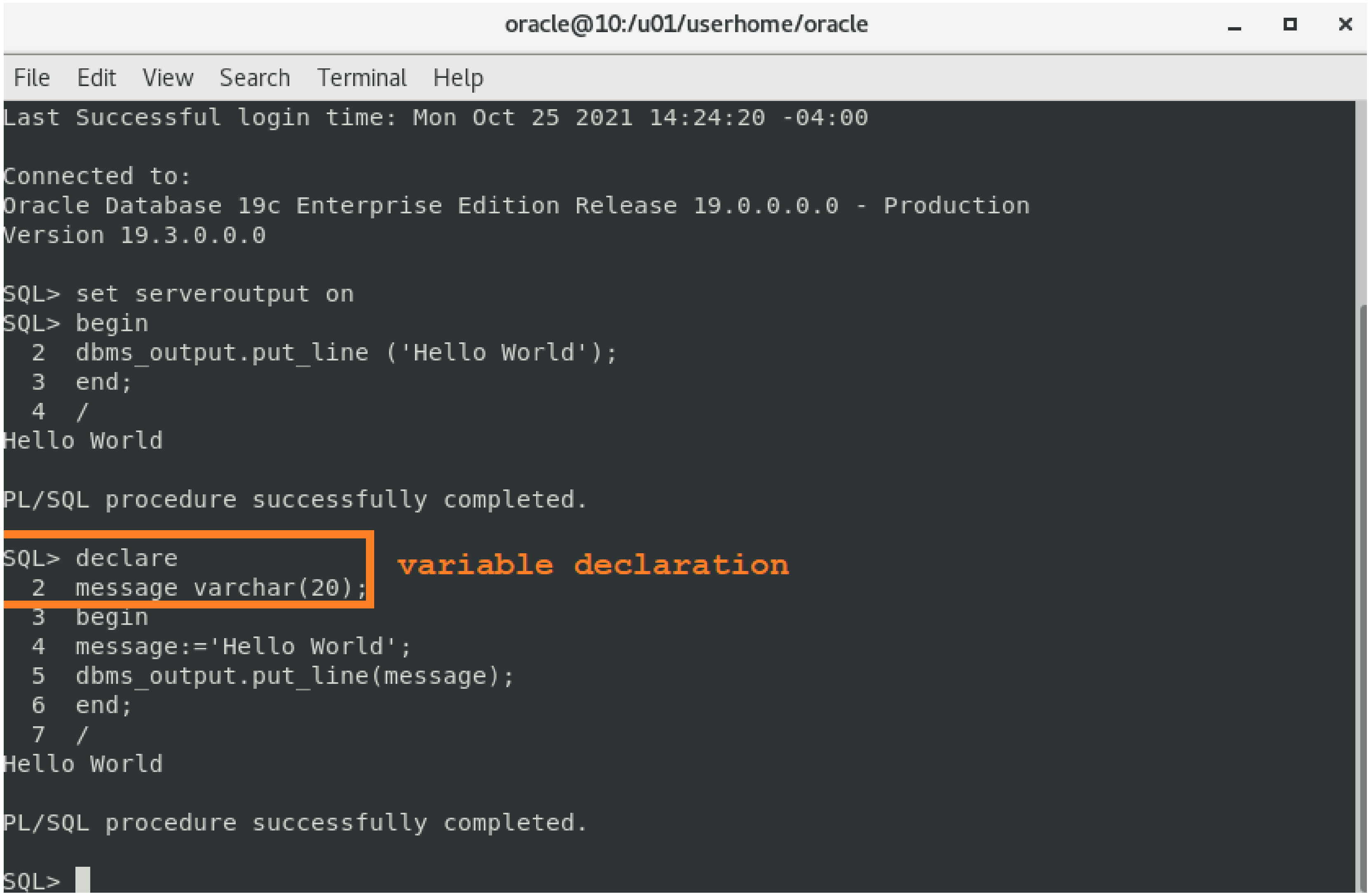
```
INSERT INTO "YELL
Commit Successful
INSERT INTO "YELL
Commit Successful
```

PL / SQL via SQL PLUS: Hello World


oracle@10:/u01/userhome/oracle
File Edit View Search Terminal Help
yellowcom
enable server output
PL/SQL code
PL/SQL output

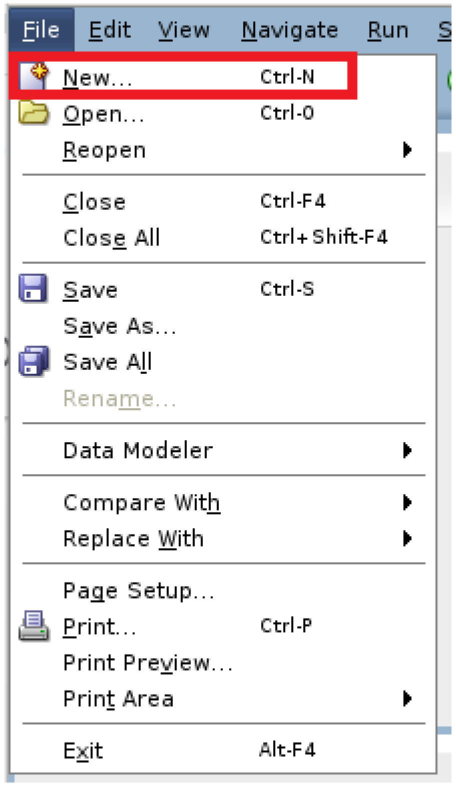
```
[oracle@10 oracle]$ sqlplus

SQL*Plus: Release 19.0.0.0.0 - Production on Mon Oct 25 14:26:38 2021
Version 19.3.0.0.0

Copyright (c) 1982, 2019, Oracle.  All rights reserved.

Enter user-name: YELLOWCOM
Enter password:
Last Successful login time: Mon Oct 25 2021 14:24:20 -04:00

Connected to:
Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production
Version 19.3.0.0.0

SQL> set serveroutput on
SQL> begin
  2  dbms_output.put_line ('Hello World');
  3  end;
  4  /
Hello World

PL/SQL procedure successfully completed.

SQL>
```

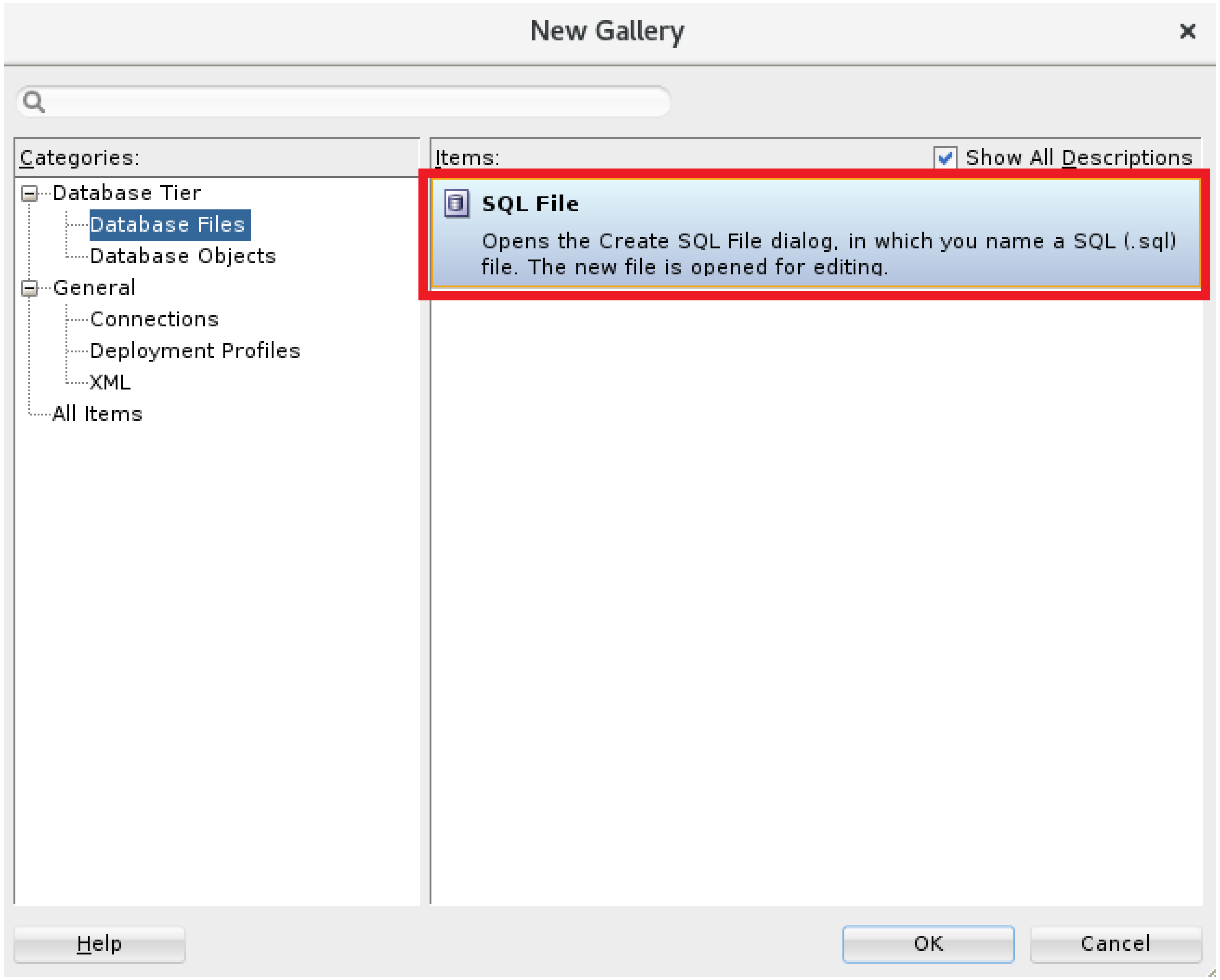
Hello World with variable


oracle@10:/u01/userhome/oracle
File Edit View Search Terminal Help
variable declaration

```
Last Successful login time: Mon Oct 25 2021 14:24:20 -04:00

Connected to:
Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production
Version 19.3.0.0.0

SQL> set serveroutput on
SQL> begin
  2  dbms_output.put_line ('Hello World');
  3  end;
  4  /
Hello World

PL/SQL procedure successfully completed.

SQL> declare
  2  message varchar(20);
  3  begin
  4  message:='Hello World';
  5  dbms_output.put_line(message);
  6  end;
  7  /
Hello World

PL/SQL procedure successfully completed.

SQL>
```

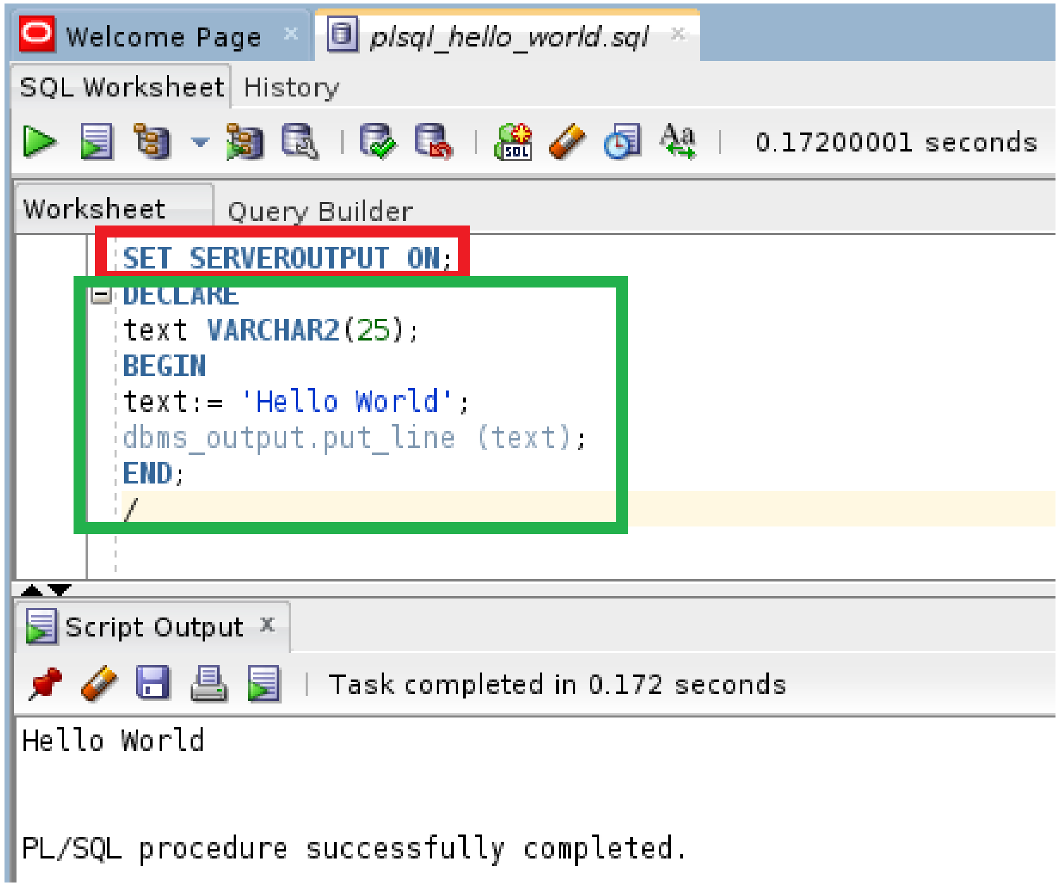
PL/SQL with SQL developer





Opens the Create SQL File dialog, in which you name a SQL (.sql) file. The new file is opened for editing.

PL/SQL with SQL developer



```
SET SERVEROUTPUT ON;
DECLARE
text VARCHAR2(25);
BEGIN
text:= 'Hello World';
dbms_output.put_line (text);
END;
/
```

```
Hello World


PL/SQL procedure successfully completed.
```

PL / SQL: the blocks ...

The PL / SQL code is grouped in ***blocks***.

A block can be:

- Without a name, in this case, is defined *anonymous*;
- A ***procedure, function*** or ***package.*** In that case, we give it a name

...PL / SQL: the blocks ...

A PL / SQL code block contains three sections:

- ***declarations***: It contains the definition and, optionally, the initialization of variables and cursors used;
- ***executable commands*** : It contains commands for flow control, such as *for*, *loop*, *if*, etc.
- ***exception handling*****:** It enables management of error conditions

...PL / SQL: the blocks ...

```
// anonymous block
DECLARE
<Section statements>
BEGIN
<Executable commands>
EXCEPTION
<Exception handling>
END;
/
```

```
// block procedure
CREATE PROCEDURE IS
<Section statements>
BEGIN
<Executable commands>
EXCEPTION
<Exception handling>
END;
/
```

...PL / SQL: the blocks

- the execution of an anonymous block takes place immediately after compiling, thus after the closure /
- the execution of a procedure block occurs at the user's request: exec procname;
- any compilation errors can be inspected with show errors;

PL / SQL: the declarations ...

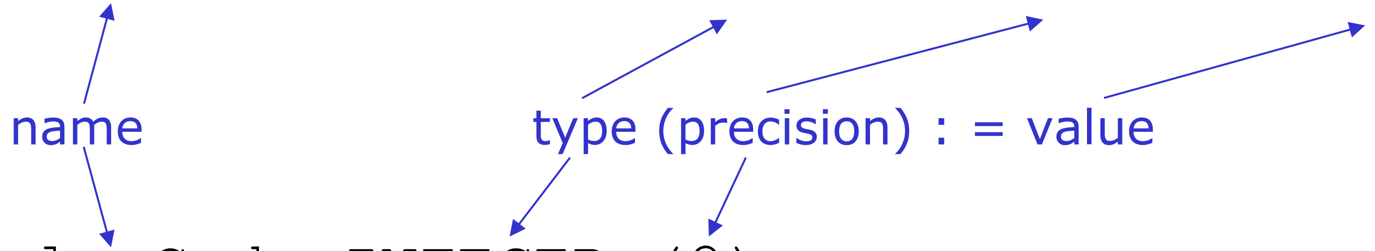
The declarations section begins with the keyword DECLARE, and starts the entire PL / SQL block.

This section contains the declaration of variables and cursors.

Variables can contain either constant values or variable values.

We can initialize them at the time of the declaration.

```
ValueCost constant NUMBER (5,2):= 180.50;
```



```
ValueCode INTEGER (2)
```

...PL / SQL: the declarations ...

The data types that can be used in PL / SQL are all valid types of SQL data.

We can also use complex types, based on query structures. For example, the following shows a cursor that is declared as a *record* from the table *Authors* and a variable whose type is based on the results of the cursor:

```
 DECLARE CURSOR author_cursor IS
    SELECT * FROM Authors;
Author author_cursor%ROWTYPE;
```

...PL / SQL: the declarations ...

We can then access the column *First name* in the following way:

```
Author.name
```

We can also declare a variable of the same type of column `Name`:

```
DECLARE varAuthor Authors.Name%TYPE
```

The use of such an inheritance of data types with definitions %ROWTYPE and %TYPE allows us to decouple the code from the underlying data structure.

...PL / SQL: the declarations ...

If the column *Name* is changed from CHAR (30) CHAR (40) we will not need to modify the PL / SQL code. The data type assigned to the variables will be determined dynamically during execution.

PL / SQL: executable commands ...

This section begins with the keyword `BEGIN`. In it cursors and variables declared in the declarations section of the PL / SQL block are processed.

In this section we can insert conditional statements and loops involving, when necessary, the cursors.

...PL / SQL: executable commands ...

- Example: With reference to the scheme:

```
Authors (Name, Surname, BirthDate, BirthPlace, ...)
Works (Code, Name, Description, ...)
```

we want to print the description of a work.

...PL / SQL: executable commands ...

// activation of the package DBMS_OUTPUT

```
SET SERVEROUTPUT ON
```

// declaration of variables and necessary sliders

```
DECLARE
   ValueCode Works.Code% TYPE;
   ValueDescription Works.description% TYPE;
CURSOR C IS
   SELECT description FROM Works
   WHERE Code = ValueCode;
```

...PL / SQL: executable commands ...

```
.
.
.
BEGIN  // body of executable commands
  ValueCode:= '111';
  OPEN c;  // open cursor
  // assignment of the column read from the cursor in the column-type
   variable
  FETCH c INTO ValueDescription;
dbms_output.put_line (ValueDescription);
  // display the value
  CLOSE c;
  // closing slider
END;
/
```

...PL / SQL: executable commands ...

The query associated with the cursor is performed when there is the call to OPEN within the block of executable commands and not at the time of declaration.

Alternatively, we could write the following without using the cursors:

```
DECLARE
     ValueCode Works.Code%TYPE;
    ValueDescription Works.Description%TYPE;
```

...PL / SQL: executable commands ...

```
BEGIN
 ValueCode: = '111';
  SELECT Description INTO ValueDescription
   FROM Works
    WHERE Code = valueCode;
 dbms_output.put_line(ValueDescription);
END;
/
```

...PL / SQL: executable commands ...

conditional logic

In the PL / SQL code we can insert a conditional logic.

Constructs *if*, *for*, *loop* and *while* are similar to those of any other programming language. The new feature is the integration with SQL statements and in particular with the cursors in order to easily access the data in the database.

...PL / SQL: executable commands ...

- **simple cycle (LOOP)**

The use of the cursor within a simple cycle provides for the opening of the cursor and the extraction of data explicitly:

loop….exit when…end loop

```
DECLARE
    ValueCode Works.Name%TYPE;
   ValueDescription Works.Description%TYPE;
CURSOR C IS
   SELECT description
   FROM Works
   WHERE name like ValueName;
```

...PL / SQL: executable commands ...

```
BEGIN
 ValueName:= 'SECOND_WORK';
 OPEN c;
 LOOP
  FETCH c INTO ValueDescription;
  EXIT WHEN c%NOTFOUND;
  dbms_output.put_line(ValueDescription);
 END LOOP;
 CLOSE c;
END;
/
```

...PL / SQL: executable commands ...

The cursors are equipped with four attributes that can be analyzed to decide, for example, when to terminate the simple cycle, as in the previous example. The four attributes are:

- %FOUND: The cursor can transmit a record;
- %NOTFOUND: The cursor can not transmit other records;
- %IsOpen: the cursor was opened;
- %ROWCOUNT: the number of lines transmitted by the cursor up to this moment.

...PL / SQL: executable commands ...

- **cycle FOR**

In a cycle *for* no cursor, the iteration is performed for a specified number of times:

for ... in ... loop ... end loop

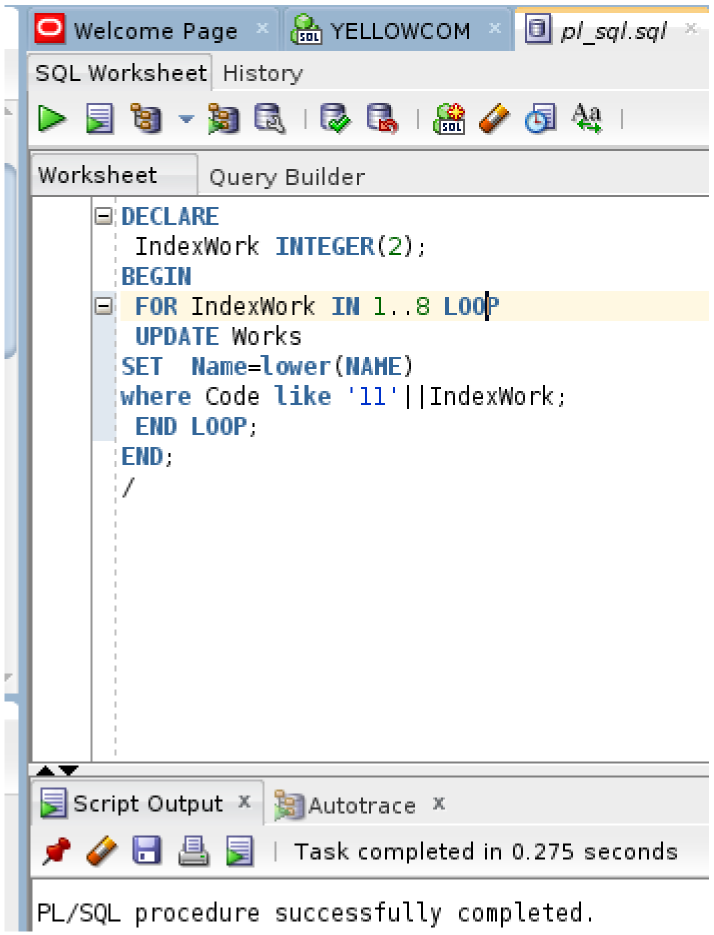
```
DECLARE
 IndexWork INTEGER(2);
BEGIN
 FOR IndexWork IN 1..8 LOOP
  UPDATE Works
  SET  Name=lower(Name)
  WHERE Code like ‘op0'||IndexWork;
 END LOOP;
END;
/
```

Note:

`||` Concatenates character strings and `CLOB` data

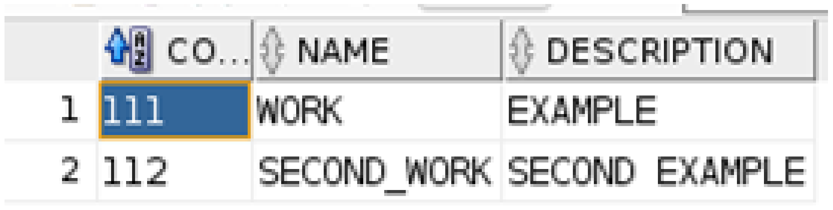
```
SELECT 'Name is ' || last_name
FROM employees;
```

...PL / SQL: executable commands ...

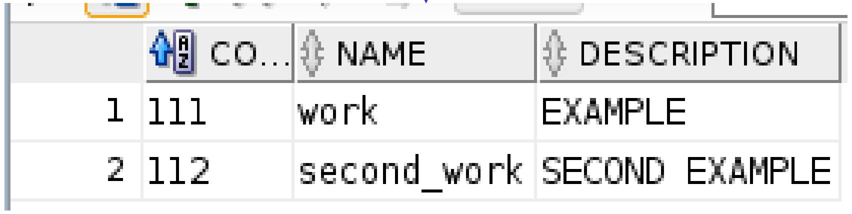


```
DECLARE
 IndexWork INTEGER(2);
BEGIN
 FOR IndexWork IN 1..8 LOOP
 UPDATE Works
SET  Name=lower(NAME)
where Code like '11'||IndexWork;
 END LOOP;
END;
/
```

```
PL/SQL procedure successfully completed.
```



	CO...	NAME	DESCRIPTION
1	111	WORK	EXAMPLE
2	112	SECOND_WORK	SECOND EXAMPLE



	CO...	NAME	DESCRIPTION
1	111	work	EXAMPLE
2	112	second_work	SECOND EXAMPLE

...PL / SQL: executable commands ...

In a *cursor-for* cycle, the results of a query are used in order to determine the number of cycles to be performed:

```
DECLARE
     CURSOR C IS SELECT * FROM Works;

BEGIN
 // pair is implicitly of type c% ROWTYPE
 FOR pair IN c LOOP
    dbms_output.put_line (pair.Name ||
                          pair.Description);
 END LOOP;
END;
/
```

...PL / SQL: executable commands ...

It should be noted that the opening of the cursor, the transmission of each line of the cursor to the variable *pair*, and finally the closing of the cursor are implicitly executed.

...PL / SQL: executable commands ...

- **WHILE loop**

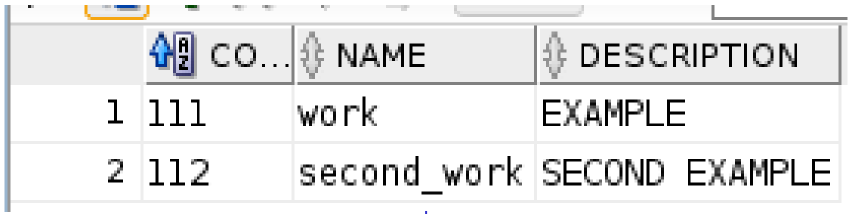
In a cycle *while,* commands are repeated until the output condition is satisfied.

while ... loop ... end loop

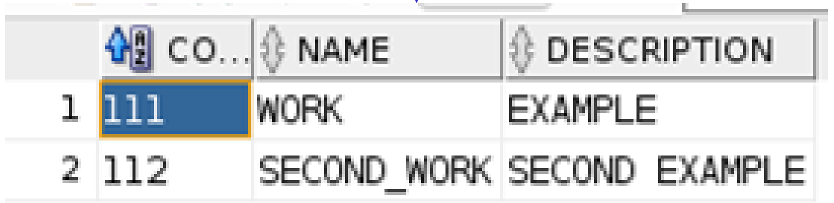
```
DECLARE
 IndexWork INTEGER (2);
BEGIN
 IndexWork: = 1;
 WHILE IndexWork <= 9 LOOP
.
.
.
```

...PL / SQL: executable commands ...

```
.
.
.
 UPDATE Works
 SET Name = upper(name)
 WHERE Code like  '11' || IndexWork;
 IndexWork:= IndexWork + 1;
END LOOP;
END;
/
```



	CO...	NAME	DESCRIPTION
1	111	work	EXAMPLE
2	112	second_work	SECOND EXAMPLE



	CO...	NAME	DESCRIPTION
1	111	WORK	EXAMPLE
2	112	SECOND_WORK	SECOND EXAMPLE

...PL / SQL: executable commands

- **GOTO**

Even in PL / SQL we can use the statement *goto* to change the flow of commands, give that we have defined labels that represents the target of the jump.

```
<<labelX>>
…
GOTO labelX;
…
```

However, we should avoid using them, given the difficult maintainability of the code and the possibility to write equivalent code, but without jumps,

PL / SQL: Exception handling ...

In the moment in which exceptions are generated by the user or by the system (errors), the control of the PL / SQL block passes to the exception section. We can use the clause *When* to determine what code to execute.

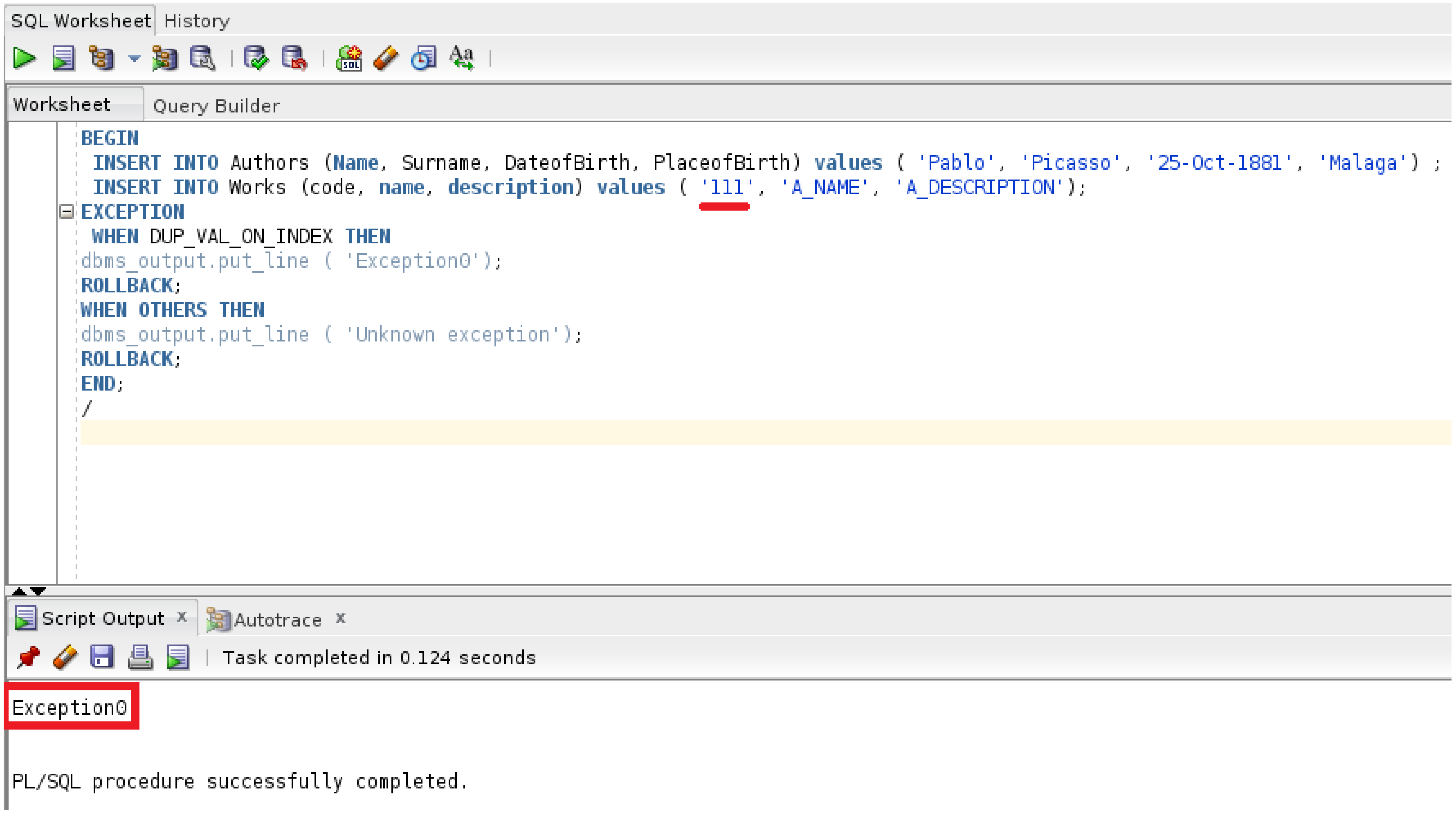
PL / SQL provides a number of exceptions defined by the system, we can also define new ones.

The section of the exceptions begins at the keyword *exception* and it is optional:

...PL / SQL: Exception handling ...

```
BEGIN
 INSERT INTO Authors (Name, Surname, DateofBirth,
  PlaceofBirth) values ( 'Pablo', 'Picasso', '25-
  Oct-1881', 'Malaga') ;
 INSERT INTO Works (code, name, description) values
  ( '111', 'A_NAME', 'A_DESCRIPTION');
EXCEPTION
 WHEN DUP_VAL_ON_INDEX THEN
   dbms_output.put_line ( 'Exception0');
   ROLLBACK;
 WHEN OTHERS THEN
   dbms_output.put_line ( 'Unknown exception');
   ROLLBACK;
END;
/
```

...PL / SQL: Exception handling ...



```
BEGIN
 INSERT INTO Authors (Name, Surname, DateofBirth, PlaceofBirth) values ( 'Pablo', 'Picasso', '25-Oct-1881', 'Malaga') ;
 INSERT INTO Works (code, name, description) values ( '111', 'A_NAME', 'A_DESCRIPTION');
EXCEPTION
 WHEN DUP_VAL_ON_INDEX THEN
dbms_output.put_line ( 'Exception0');
ROLLBACK;
WHEN OTHERS THEN
dbms_output.put_line ( 'Unknown exception');
ROLLBACK;
END;
/
```

Task completed in 0.124 seconds

```
Exception0


PL/SQL procedure successfully completed.
```

...PL / SQL: Exception handling ...

When an error happens, the DBMS searches, in the exceptions section, the code for the management of the specific exception.

Apart from the exceptions defined by the system and those defined by the user, we can insert the clause `WHEN OTHERS` to catch all exceptions not caught by other clauses `WHEN`.

...PL / SQL: exception handling

Note that we can not return to normal flow control after handling the exception.

Even the GOTO statement used for that purpose fails. In the case where it is necessary not to go out from the normal flow of control we will have to prevent the generation of exceptions, through the use of conditional statements, rather than manage them after they have occurred.

PL / SQL: Procedures and Functions ...

We conclude this brief overview of PL / SQL noting that we can define procedures, functions, and modules that can be stored in the database, by assigning them a name (CREATE PROCEDURE, CREATE PACKAGE, CREATE FUNCTION).

Procedures and modules stored can be recalled and then used by other programs, even remotely.

...PL / SQL: Procedures and Functions ...

consider an anonymous block previously seen:

```
DECLARE
   ValueCode Works.Name% TYPE;
   ValueDescription Works.Description%TYPE;
   CURSOR C IS
     SELECT description FROM Works
   WHERE name like ValueName;
BEGIN
 ValueName: = 'daphne and apollo';
 OPEN c;
 LOOP
  FETCH c INTO ValueDescription;
  EXIT WHEN c%NOTFOUND;
  dbms_output.put_line(ValueDescription);
 END LOOP;
 CLOSE c;
END;
/
```

...PL / SQL: Procedures and Functions ...

if we want to define a procedure:

```
CREATE PROCEDURE PRINTDESCRIPTION IS
  ValueName Works.Name%TYPE;
  ValueDescription Works.Description%TYPE;
  CURSOR C IS
    SELECT description FROM Works
    WHERE name like ValueName;
 BEGIN
  ValueName:='dafne e apollo';
  OPEN c;
  LOOP
   FETCH c INTO ValueDescription;
   EXIT WHEN c%NOTFOUND;
   dbms_output.put_line(ValueDescription);
  END LOOP;
  CLOSE c;
 END;
    /
```

...PL / SQL: Procedures and Functions ...

if we want to define a procedure:

```
CREATE PROCEDURE PRINTDESCRIPTION IS
   ValueName Works.Name%TYPE;
   ValueDescription Works.Description%TYPE;
   CURSOR C IS
     SELECT description FROM Works
     WHERE name like ValueName;
 BEGIN
   ValueName:='dafne e apollo';
   OPEN c;
   LOOP
    FETCH c INTO ValueDescription;
    EXIT WHEN c%NOTFOUND;
    dbms_output.put_line(ValueDescription);
   END LOOP;
   CLOSE c;
 END;
     /
```

-> exec PROC (*parameter_value*);

...PL / SQL: Procedures and Functions ...

if you want to define a parameterized function:

```
CREATE FUNCTION GIVEDESCRIPTION (ValueCode
                Works.Code%TYPE)
                RETURN Works.Description%TYPE AS
  ValueDescription Works.Description%TYPE;
  BEGIN
     SELECT Description  INTO ValueDescription
     FROM Works
     WHERE ID = ValueCode;
     return ValueDescription;
  END;
  /
```

...PL / SQL: Procedures and Functions

To test a function, we can use a query on a dummy table called "DUAL":

```
SELECT GIVEDESCRIPTION ( '111') FROM DUAL;
```

The table "DUAL" does not exist, it is a convenient name that simulates a single-row table.