

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
TECHNICAL UNIVERSITY OF MOLDOVA

DATABASES II
COURSE PROJECT

Control Structures in MSSQL and Oracle

Authors:

Oxana DUNAV

Supervisor:

Irina COJANU

Contents

Introduction	3
1 Control Structures in PL/SQL	4
1.1 Conditional Control: IF and CASE Statements	4
1.1.1 IF-THEN	4
1.1.2 IF-THEN-ELSE	4
1.1.3 IF-THEN-ELSIF	5
1.1.4 CASE Statement	6
1.1.5 Searched CASE Statement	6
1.2 Iterative Control: LOOP and EXIT Statements	7
1.2.1 LOOP	7
1.2.2 EXIT	7
1.2.3 EXIT-WHEN	7
1.2.4 WHILE-LOOP	8
1.2.5 FOR-LOOP	8
1.3 Sequential Control: GOTO and NULL Statements	9
1.3.1 GOTO Statement	9
1.3.2 NULL Statement	10
2 Control Structures in T-SQL	11
2.1 IF...ELSE	11
2.2 CASE	12
2.3 WHILE-BREAK-CONTINUE	14
2.4 GOTO	14
3 Practical Part	16
3.1 Differences and similarities between MSSQL and Oracle	16
Conclusions	22
References	23

Introduction

A control structure is a block of programming that analyzes variables and chooses a direction in which to go based on given parameters. The term *flow control* details the direction the program takes (which way program control "flows"). Hence it is the basic decision-making process in computing; flow control determines how a computer will respond when given certain conditions and parameters.

Those initial conditions and parameters are called preconditions. Preconditions are the state of variables before entering a control structure. Based on those preconditions, the computer runs an algorithm (the control structure) to determine what to do. The result is called a postcondition. Postconditions are the state of variables after the algorithm is run.

According to the structure theorem, any computer program can be written using the basic control structures shown in Figure 1. They can be combined in any way necessary to deal with a given problem.

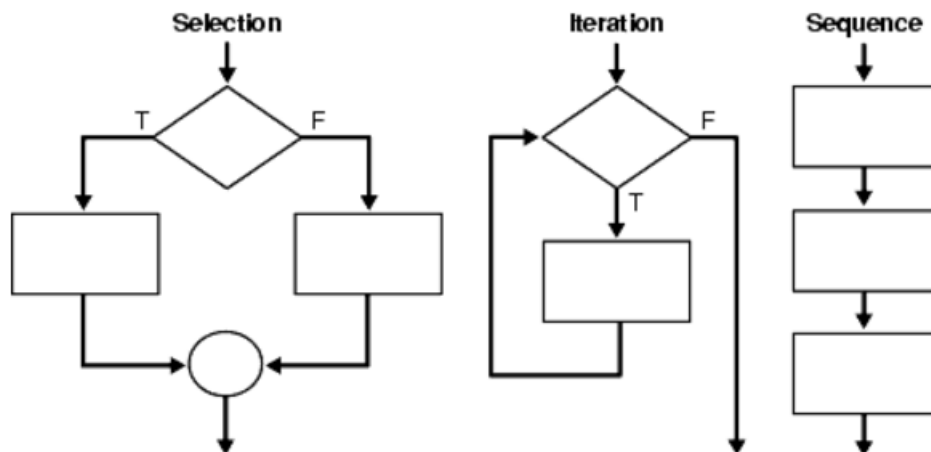


Figure 0.1 – Control Structures

The *selection* structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a Boolean value (TRUE or FALSE). The *iteration* structure executes a sequence of statements repeatedly as long as a condition holds true. The *sequence* structure simply executes a sequence of statements in the order in which they occur.

1 Control Structures in PL/SQL

1.1 Conditional Control: IF and CASE Statements

The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions.

1.1.1 IF-THEN

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

```
IF condition THEN
    sequence_of_statements
END IF;
```

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement. An example follows:

```
1 IF sales > quota THEN
2     compute_bonus(empid);
3     UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
4 END IF;
```

1.1.2 IF-THEN-ELSE

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
    sequence_of_statements1
ELSE
    sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed. In the following example, the first UPDATE statement is executed when the condition is true, but the second UPDATE statement is executed when the condition is false or null:

```
1 IF trans_type = 'CR' THEN
2     UPDATE accounts SET balance = balance + credit WHERE ...
3 ELSE
```

```

4 UPDATE accounts SET balance = balance - debit WHERE ...
5 END IF;

```

The THEN and ELSE clauses can include IF statements. That is, IF statements can be nested, as the following example shows:

```

1 IF trans_type = 'CR' THEN
2 UPDATE accounts SET balance = balance + credit WHERE ...
3 ELSE
4 IF new_balance >= minimum_balance THEN
5 UPDATE accounts SET balance = balance - debit WHERE ...
6 ELSE
7 RAISE insufficient_funds;
8 END IF;
9 END IF;

```

1.1.3 IF-THEN-ELSIF

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

```

IF condition1 THEN
    sequence_of_statements1
ELSIF condition2 THEN
    sequence_of_statements2
ELSE
    sequence_of_statements3
END IF;

```

If the first condition is false or null, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is true, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or null, the sequence in the ELSE clause is executed. Consider the following example:

```

1 BEGIN
2 ...
3 IF sales > 50000 THEN
4 bonus := 1500;
5 ELSIF sales > 35000 THEN
6 bonus := 500;
7 ELSE
8 bonus := 100;
9 END IF;
10 INSERT INTO payroll VALUES (emp_id, bonus, ...);

```

```
11 END;
```

If the value of sales is larger than 50000, the first and second conditions are true. Nevertheless, bonus is assigned the proper value of 1500 because the second condition is never tested. When the first condition is true, its associated statement is executed and control passes to the INSERT statement.

1.1.4 CASE Statement

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. The CASE statement has the following form:

CASE selector

```
    WHEN expression1 THEN sequence_of_statements1;
    WHEN expression2 THEN sequence_of_statements2;
    ...
    WHEN expressionN THEN sequence_of_statementsN;
    [ELSE sequence_of_statementsN+1;]
END CASE;
```

1.1.5 Searched CASE Statement

PL/SQL also provides a searched CASE statement, which has the form:

[<<label_name>>]

CASE

```
    WHEN search_condition1 THEN sequence_of_statements1;
    WHEN search_condition2 THEN sequence_of_statements2;
    ...
    WHEN search_conditionN THEN sequence_of_statementsN;
    [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

The searched CASE statement has no selector. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type. An example follows:

```
1 CASE
2   WHEN grade = 'A' THEN dbms_output.put_line('Excellent');
3   WHEN grade = 'B' THEN dbms_output.put_line('Very Good');
4   WHEN grade = 'C' THEN dbms_output.put_line('Good');
5   WHEN grade = 'D' THEN dbms_output.put_line('Fair');
6   WHEN grade = 'F' THEN dbms_output.put_line('Poor');
7   ELSE dbms_output.put_line('No such grade');
8 END CASE;
```

The search conditions are evaluated sequentially. The Boolean value of each search condition determines which WHEN clause is executed. If a search condition yields TRUE, its WHEN clause is executed. If any WHEN clause is executed, control passes to the next statement, so subsequent search conditions are not evaluated.

1.2 Iterative Control: LOOP and EXIT Statements

1.2.1 LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
    sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

1.2.2 EXIT

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
1 LOOP
2     ...
3     IF credit_rating < 3 THEN
4         ...
5         EXIT;  -- exit loop immediately
6     END IF;
7 END LOOP;
8 -- control resumes here
```

1.2.3 EXIT-WHEN

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop. An example follows:

```
1 LOOP
2     FETCH c1 INTO ...
3     EXIT WHEN c1%NOTFOUND;  -- exit loop if condition is true
```

```
4    ...
5 END LOOP;
6 CLOSE c1;
```

Until the condition is true, the loop cannot complete. So, a statement inside the loop must change the value of the condition. In the last example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.

1.2.4 WHILE-LOOP

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
    sequence_of_statements
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement. An example follows:

```
1 WHILE total <= 25000 LOOP
2     ...
3     SELECT sal INTO salary FROM emp WHERE ...
4     total := total + salary;
5 END LOOP;
```

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times. In the last example, if the initial value of total is larger than 25000, the condition is false and the loop is bypassed.

1.2.5 FOR-LOOP

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax follows:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
    sequence_of_statements
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated.

As the next example shows, the sequence of statements is executed once for each integer in the range. After each iteration, the loop counter is incremented.


```

1 FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
2     sequence_of_statements -- executes three times
3 END LOOP;

```

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword `REVERSE`, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. Nevertheless, you write the range bounds in ascending (not descending) order.

```

1 FOR i IN REVERSE 1..3 LOOP -- assign the values 3,2,1 to i
2     sequence_of_statements -- executes three times
3 END LOOP;

```

1.3 Sequential Control: GOTO and NULL Statements

Unlike the `IF` and `LOOP` statements, the `GOTO` and `NULL` statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the `GOTO` statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The `NULL` statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of `GOTO` statements can result in complex, unstructured code (sometimes called spaghetti code) that is hard to understand and maintain. So, use `GOTO` statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a `GOTO` statement.

1.3.1 GOTO Statement

The `GOTO` statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the `GOTO` statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```

1 BEGIN
2     ...
3     GOTO insert_row;
4     ...
5     <<insert_row>>
6     INSERT INTO emp VALUES ...
7 END;

```

In the next example, you go to a PL/SQL block farther up in a sequence of statements:

```

1 BEGIN
2     ...
3     <<update_row>>
4     BEGIN

```

```

5      UPDATE emp SET ...
6      ...
7  END;
8      ...
9  GOTO update_row;
10     ...
11 END;

```

1.3.2 NULL Statement

The NULL statement does nothing other than pass control to the next statement. In a conditional construct, the NULL statement tells readers that a possibility has been considered, but no action is necessary. In the following example, the NULL statement shows that no action is taken for unnamed exceptions:

```

1 EXCEPTION
2   WHEN ZERO_DIVIDE THEN
3     ROLLBACK;
4   WHEN VALUE_ERROR THEN
5     INSERT INTO errors VALUES ...
6     COMMIT;
7   WHEN OTHERS THEN
8     NULL;
9 END;

```

In IF statements or other places that require at least one executable statement, the NULL statement to satisfy the syntax. In the following example, the NULL statement emphasizes that only top-rated employees get bonuses:

```

1 IF rating > 90 THEN
2   compute_bonus(emp_id);
3 ELSE
4   NULL;
5 END IF;

```

Also, the NULL statement is a handy way to create stubs when designing applications from the top down. A stub is dummy subprogram that lets you defer the definition of a procedure or function until you test and debug the main program. In the following example, the NULL statement meets the requirement that at least one statement must appear in the executable part of a subprogram:

```

1 PROCEDURE debit_account (acct_id INTEGER, amount REAL) IS
2 BEGIN
3   NULL;
4 END debit_account;

```

2 Control Structures in T-SQL

The Transact-SQL control-of-flow language keywords for are:

- BEGIN...END
- BREAK
- CONTINUE
- GOTO label
- RETURN
- IF...ELSE
- THROW
- TRY...CATCH
- WAITFOR
- WHILE.

2.1 IF...ELSE

Imposes conditions on the execution of a Transact-SQL statement. The Transact-SQL statement that follows an IF keyword and its condition is executed if the condition is satisfied: the Boolean expression returns TRUE. The optional ELSE keyword introduces another Transact-SQL statement that is executed when the IF condition is not satisfied: the Boolean expression returns FALSE.

Syntax:

```
IF Boolean_expression
    { sql_statement | statement_block }
[ ELSE
    { sql_statement | statement_block } ]
```

Arguments:

Boolean_expression

Is an expression that returns TRUE or FALSE. If the Boolean expression contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

sql_statement| statement_block

Is any Transact-SQL statement or statement grouping as defined by using a statement block. Unless a statement block is used, the IF or ELSE condition can affect the performance of only one Transact-SQL statement. To define a statement block, use the control-of-flow keywords BEGIN and END.

Example:

```
1 IF DATENAME(weekday, GETDATE()) IN (N'Saturday', N'Sunday')
2     SELECT 'Weekend';
3 ELSE
4     SELECT 'Weekday';
```

Using nested IF...ELSE statements

The following example shows how an IF ... ELSE statement can be nested inside another. Set the @Number variable to 5, 50, and 500 to test each statement.

```
1 DECLARE @Number int;
2 SET @Number = 50;
3 IF @Number > 100
4     PRINT 'The number is large.';
5 ELSE
6     BEGIN
7         IF @Number < 10
8             PRINT 'The number is small.';
9         ELSE
10            PRINT 'The number is medium.';
11    END ;
12 GO
```

2.2 CASE

Evaluates a list of conditions and returns one of multiple possible result expressions. The CASE expression has two formats:

- The simple CASE expression compares an expression to a set of simple expressions to determine the result.
- The searched CASE expression evaluates a set of Boolean expressions to determine the result.

Both formats support an optional ELSE argument. CASE can be used in any statement or clause that allows a valid expression. For example, you can use CASE in statements such as SELECT, UPDATE, DELETE and SET, and in clauses such as select_list, IN, WHERE, ORDER BY, and HAVING.

Syntax:

Simple CASE expression:

```
CASE input_expression
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Searched CASE expression:

CASE

```
    WHEN Boolean_expression THEN result_expression [ ...n ]  
    [ ELSE else_result_expression ]
```

END

Example:

- Using a SELECT statement with a simple CASE expression Within a SELECT statement, a simple CASE expression allows for only an equality check; no other comparisons are made. The following example uses the CASE expression to change the display of product line categories to make them more understandable.

```
1 USE AdventureWorks2012;  
2 GO  
3 SELECT    ProductNumber, Category =  
4           CASE ProductLine  
5             WHEN 'R' THEN 'Road'  
6             WHEN 'M' THEN 'Mountain'  
7             WHEN 'T' THEN 'Touring'  
8             WHEN 'S' THEN 'Other sale items'  
9             ELSE 'Not for sale'  
10          END,  
11          Name  
12 FROM Production.Product  
13 ORDER BY ProductNumber;  
14 GO
```

- Using a SELECT statement with a searched CASE expression Within a SELECT statement, the searched CASE expression allows for values to be replaced in the result set based on comparison values. The following example displays the list price as a text comment based on the price range for a product.

```
1 USE AdventureWorks2012;  
2 GO  
3 SELECT    ProductNumber, Name, "Price Range" =  
4           CASE  
5             WHEN ListPrice = 0 THEN 'Mfg item - not for resale'  
6             WHEN ListPrice < 50 THEN 'Under $50'  
7             WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'  
8             WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'  
9             ELSE 'Over $1000'  
10          END  
11 FROM Production.Product  
12 ORDER BY ProductNumber ;  
13 GO
```

2.3 WHILE-BREAK-CONTINUE

Sets a condition for the repeated execution of an SQL statement or statement block. The statements are executed repeatedly as long as the specified condition is true. The execution of statements in the WHILE loop can be controlled from inside the loop with the BREAK and CONTINUE keywords.

Syntax:

```
WHILE Boolean_expression
    { sql_statement | statement_block | BREAK | CONTINUE }
```

BREAK

Causes an exit from the innermost WHILE loop. Any statements that appear after the END keyword, marking the end of the loop, are executed.

CONTINUE

Causes the WHILE loop to restart, ignoring any statements after the CONTINUE keyword.

Example:

Using BREAK and CONTINUE with nested IF...ELSE and WHILE:

In the following example, if the average list price of a product is less than 300, the WHILE loop doubles the prices and then selects the maximum price. If the maximum price is less than or equal to 500, the WHILE loop restarts and doubles the prices again. This loop continues doubling the prices until the maximum price is greater than 500, and then exits the WHILE loop and prints a message.

```
1 USE AdventureWorks2012;
2 GO
3 WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
4 BEGIN
5     UPDATE Production.Product
6         SET ListPrice = ListPrice * 2
7     SELECT MAX(ListPrice) FROM Production.Product
8     IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
9         BREAK
10    ELSE
11        CONTINUE
12 END
13 PRINT 'Too much for the market to bear';
```

2.4 GOTO

Alters the flow of execution to a label. The Transact-SQL statement or statements that follow GOTO are skipped and processing continues at the label. GOTO statements and labels can be used anywhere within a procedure, batch, or statement block. GOTO statements can be nested.

Syntax:

Define the label:

label:

Alter the execution:

GOTO label

label

Is the point after which processing starts if a GOTO is targeted to that label. Labels must follow the rules for identifiers. A label can be used as a commenting method whether GOTO is used.

Example:

The following example shows how to use GOTO as a branch mechanism.

```
1 DECLARE @Counter int;
2 SET @Counter = 1;
3 WHILE @Counter < 10
4 BEGIN
5     SELECT @Counter
6     SET @Counter = @Counter + 1
7     IF @Counter = 4 GOTO Branch_One --Jumps to the first branch.
8     IF @Counter = 5 GOTO Branch_Two --This will never execute.
9 END
10 Branch_One:
11     SELECT 'Jumping To Branch One.'
12     GOTO Branch_Three; --This will prevent Branch_Two from executing.
13 Branch_Two:
14     SELECT 'Jumping To Branch Two.'
15 Branch_Three:
16     SELECT 'Jumping To Branch Three.';
```

3 Practical Part

3.1 Differences and similarities between MSSQL and Oracle

The control structures have the same logic when comes to determine how the computer responds to certain conditions and parameters. These applies not only to SQL, but to other programming languages as C, C++, Python, Java etc.

In this chapter we will analyze how certain examples in MSSQL and Oracle. The examples will be written for the database calculatoare.

– IF Statement

The first difference is the syntax used for the statement: In Oracles we should use the keyword *THEN* after IF and ends the statement with the keyword *END IF* whereas in MSSQL it is not needed.

The second difference is that in MSSQL we can use a query as part of a Boolean expression, but in Oracle we can not do this, so instead we can declare a variable that will hold the value of the query and after that operate on it.

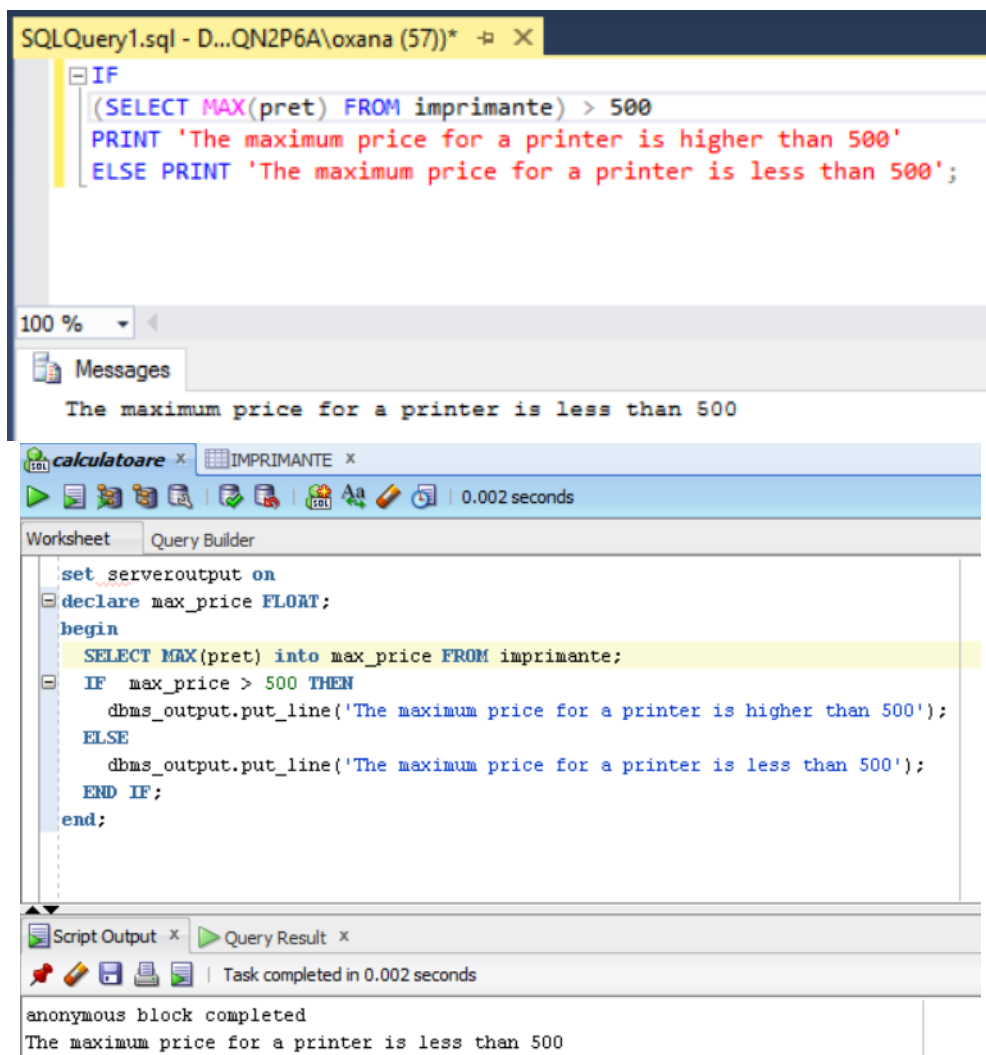


Figure 3.1– IF statement in MSSQL and Oracle

The maximum price from the printers is 300. So, in this case it is executed the condition after

ELSE.

As for the similarities, both MSSQL and Oracle allow nested conditions.

– CASE Statement

The difference is that in Oracle if you omit the ELSE clause and none of the search conditions before yields TRUE, than PL/SQL adds the following implicit ELSE clause: ELSE RAISE CASE_NOT_FOUND;

In MSSQL, when this argument is omitted and no comparison operation evaluates to TRUE, CASE returns NULL.

The similarity is that in both MSSQL and Oracle can be used searched CASE expressions.

Here is a simple case statement, that compares the selector with the values after WHEN:

The top screenshot shows a SQL Developer window with a query in 'Untitled2.sql' and its results. The query is:

```
SELECT Cod, Model,
CASE Pret
WHEN 0 THEN 'not for sale'
WHEN 50 THEN '$50'
WHEN 170 THEN '$170'
WHEN 300 THEN '$300'
ELSE 'not known'
END as PriceRange
FROM imprimante;
```

The results table has 6 rows:

	COD	MODEL	PRICERANGE
1	1	1276	\$300
2	2	1433	\$170
3	3	1434	not known
4	4	1401	\$50
5	5	1408	\$170
6	6	1288	\$300

The bottom screenshot shows a SQL Developer window with a query in 'SQLQuery3.sql' and its results. The query is:

```
SELECT Cod, Model,
CASE Pret
WHEN 0 THEN 'not for sale'
WHEN 150 THEN '$170'
WHEN 270 THEN '$240'
WHEN 400 THEN '$400'
ELSE 'not known'
END as PretRange
FROM copiatoare.imprimante;
```

The results table has 7 rows:

	Cod	Model	PretRange
1	1	1276	\$400
2	2	1433	\$240
3	3	1434	not known
4	4	1401	\$170
5	5	1408	\$240
6	6	1288	\$400
7	7	1290	not known

Figure 3.2– Case statement in MSSQL and Oracle

Now, we can show a searched case, which uses expressions:

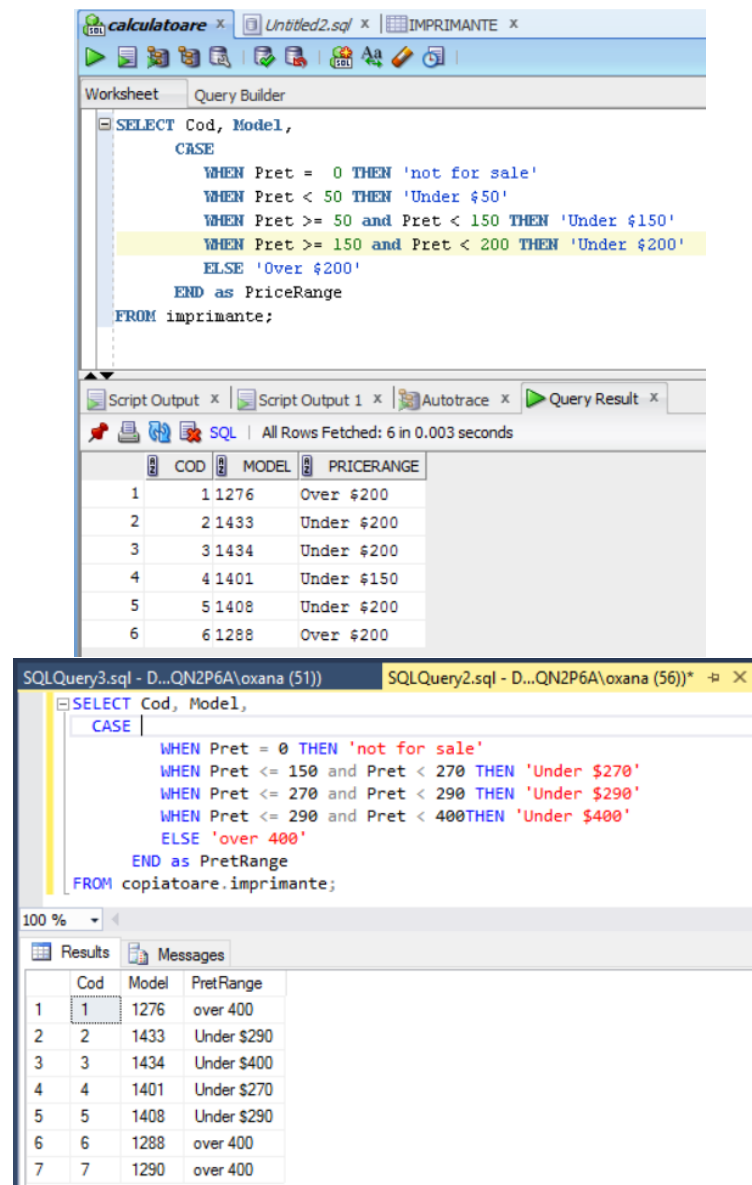


Figure 3.3– Searched case statement in MSSQL and Oracle

– Iterative Control

The first main difference is that in Oracle we have both WHILE and FOR loops, but in MSSQL there is no FOR loop. In MSSQL we simulate the FOR LOOP using the WHILE LOOP.

Another difference is that in MSSQL you can exit a while loop with the keyword BREAK, but in Oracle with the keyword EXIT or EXIT-WHEN.

Here is an While loop, that can break or continue using the specified keywords. In the following example, if the average price of a printer is less than \$300, the WHILE loop doubles the prices and then selects the maximum price. If the maximum price is less than or equal to \$500, the WHILE loop restarts and doubles the prices again. This loop continues doubling the prices until the maximum price is greater than \$500, and then exits the WHILE loop and prints a message.

The screenshot shows the MSSQL Enterprise interface. The top bar displays two tabs: 'SQLQuery3.sql - D...QN2P6A\oxana (51))' and 'SQLQuery2.sql - D...QN2P6A\oxana (56))*'. The main editor contains the following SQL code:

```

WHILE (SELECT AVG(Pret) FROM copiatoare.imprimante) < 300
BEGIN
    UPDATE copiatoare.imprimante
    SET Pret = Pret * 2
    SELECT MAX(Pret) FROM copiatoare.imprimante
    IF (SELECT MAX(Pret) FROM copiatoare.imprimante) > 500
        BREAK
    ELSE
        CONTINUE
END
PRINT 'Too much for the market to bear';

```

Below the editor, a 'Messages' pane shows the output: 'Too much for the market to bear'.

Figure 3.4– While loop in MSSQL

In Oracle it is also used the LOOP keyword. In the following case the avg_price will be doubled until it gets bigger than 400.

The screenshot shows the Oracle SQL Developer interface. The top bar includes tabs for 'calculatoare', 'Untitled2.sql', and 'IMPRIMANTE'. The main editor displays the following PL/SQL code:

```

declare avg_price FLOAT;
begin
    SELECT AVG(pret) into avg_price from imprimante;
    WHILE avg_price < 400
    LOOP
        avg_price := avg_price * 2;
    END LOOP;
end;

```

The bottom pane shows the 'Script Output' tab with the message: 'anonymous block completed'.

Figure 3.5– While loop in Oracle

In Oracle, the FOR LOOP allows you to execute code repeatedly for a fixed number of times. This FOR LOOP example will loop 20 times. The counter called i will start at 1 and end at 20. So, it will double the avg-price 20 times.

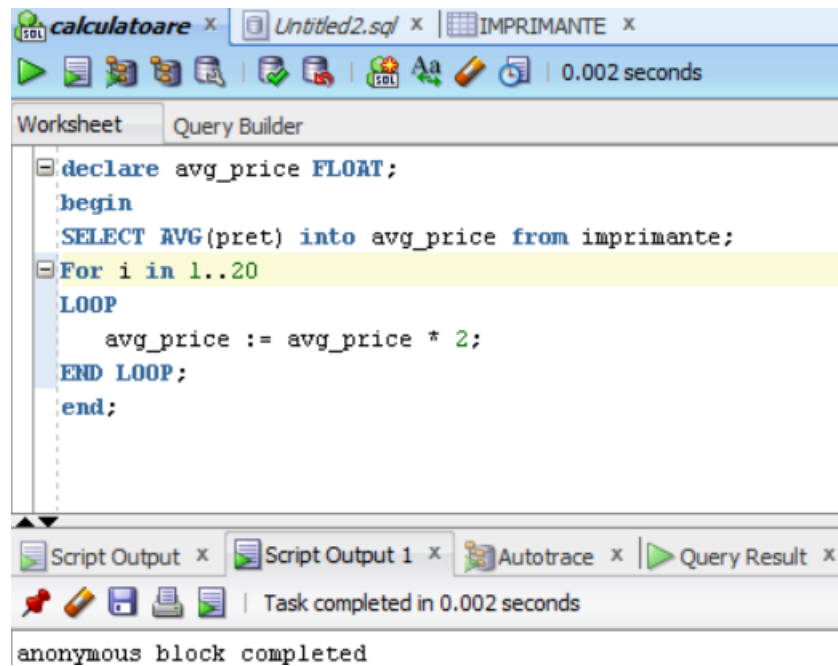


Figure 3.6 – For loop in Oracle

– GOTO

Both in Oracle and MSSQL use GOTO label to alter the flow of the execution. They have similar implementation, the only difference is that in Oracle the label name is in between \ll and \gg .

The following example in MSSQL will jump to the finished label when the maximum price of the printer will exceed 300.

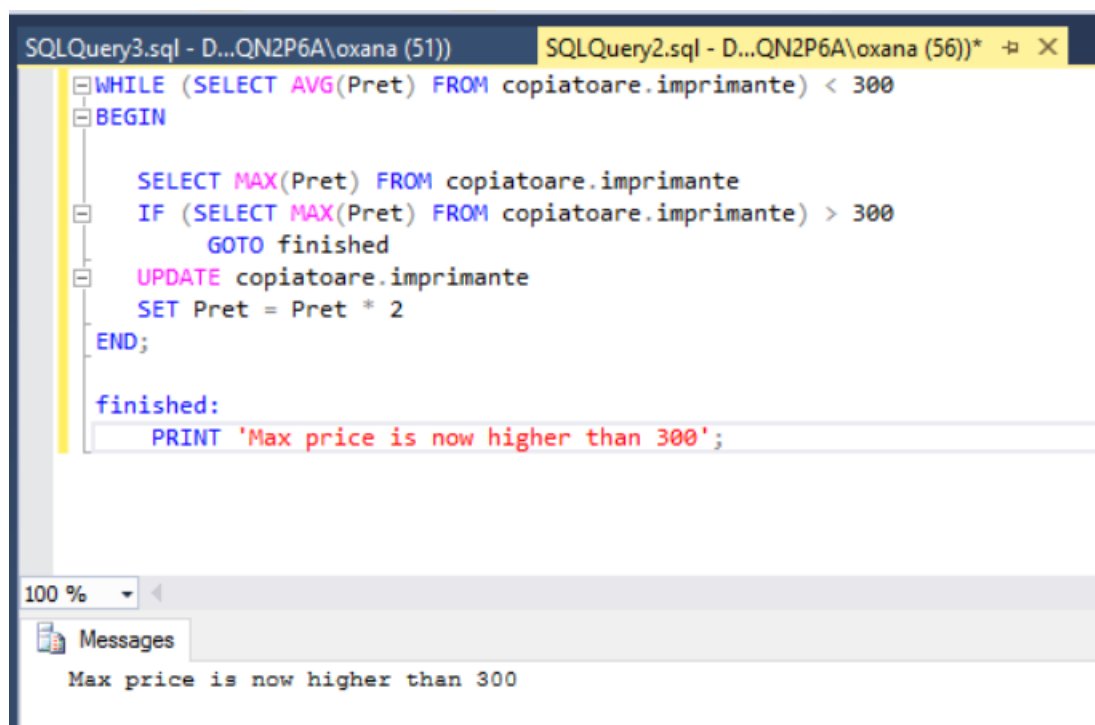
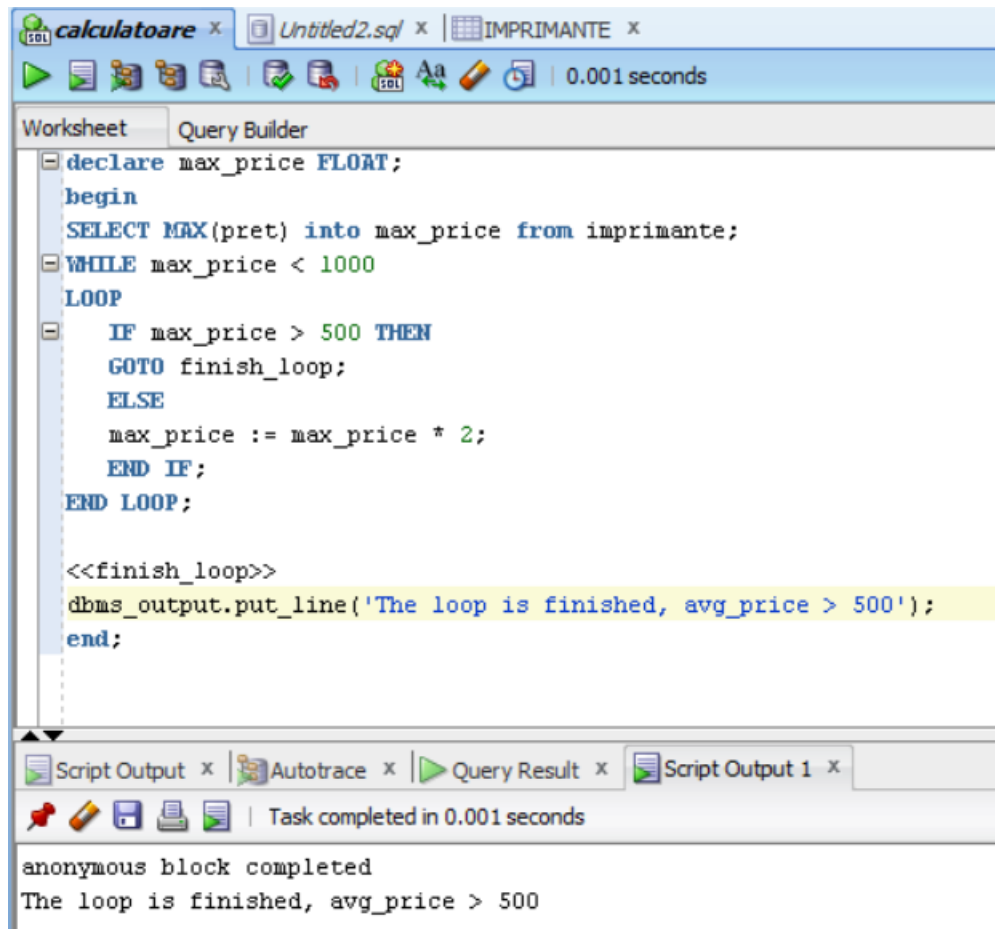


Figure 3.7 – GOTO in MSSQL

The next example, in Oracle has a while loop that will stop when the max price will be higher than 1000. However, the goto label can interrupt the flow of control and jump to the printing line. So, in this case when the max price gets higher than 500, it will jump to the label finish_loop.



The screenshot shows the SQL Developer interface with a script editor containing the following PL/SQL code:

```
declare max_price FLOAT;
begin
  SELECT MAX(pret) into max_price from imprimante;
  WHILE max_price < 1000
  LOOP
    IF max_price > 500 THEN
      GOTO finish_loop;
    ELSE
      max_price := max_price * 2;
    END IF;
  END LOOP;

  <<finish_loop>>
  dbms_output.put_line('The loop is finished, avg_price > 500');
end;
```

The script is executed, and the output pane at the bottom shows the results:

```
anonymous block completed
The loop is finished, avg_price > 500
```

Figure 3.8– GOTO in Oracle

Conclusions

For the course project I have researched the control structures in PL/SQL and T-SQL and for the practice part I have used Oracle and Microsoft SQL Server.

First of all, there are three types of control structures: conditional selection statements, which run different statements for different data values(If and Case), loop statements, which run the same statements with a series of different data values(FOR, While) and sequential control statements, like GOTO, which goes to a specified statement, and NULL, which does nothing.

These have the same logic for either Oracle and MSSQL and other programming languages as well. But between Oracle and MSSQL there are some differences when it comes to their implementation and it is not only about the syntax. For example, there are some keywords that are used only in MSSQL, like BREAK and CONTINUE whereas in Oracle it is used the EXIT and EXIT WHEN keywords. Also, one main difference is in the iteration part - in MSSQL there is no FOR loop and you can simulate it using a while loop instead.

During this course project, I have learned each structure for both MSSQL and Oracle and then I have deduced the similarities and the differences for them. I have created several example in Oracle and MSSQL in order to show how they are implemented and to see the result when executing them.

I think that control structures are a very powerful tool which allow developers to do much more with a structured and logical source code. That's why it is important to understand the concept of control structures, to know how to use them and when it is necessary to use. When you understand them, it is easier to use either in Oracle or in MSSQL. Using the documentation that they provide, we can easily find the syntax and some specific rules for the context used.

Personally, I have got a lot of knowledge doing this course project and when doing the practical part, it also has helped me in understanding better how to work in Oracle as well as in SQL. This kind of course projects not only that teaches you some information and how to implement it, but Besides this, it teaches how to research an unknown subject, to use the documentation from the official parts and to structure everything in a report.

References

- 1 Oracle, *Oracle PL/SQL documentation* https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/04_struct.htm I
- 2 Microsoft, *Microsoft SQL documentation* <https://docs.microsoft.com/en-us/sql/t-sql/language-elements/goto-transact-sql>
- 3 Oracle Tutorial, *Oracle Tutorial* <https://www.techonthenet.com/oracle/index.php>
- 4 SQL Server Tutorial, *SQL Server Tutorial* https://www.techonthenet.com/sql_server/index.php