4 Writing Control Structures

✅Identify the uses and types of control structures (IF, CASE statements and expressions)

Types of control statements, IF, CASE: 4 - 4.1.5

PL/SQL has three categories of control statements: conditional selection statements, loop statements and sequential control statements. PL/SQL categories of control statements are:

- **conditional selection statements**, which run different statements for different data values. The conditional selection statements are `IF` and `CASE`
- **loop statements**, which run the same statements with a series of different data values. The loop statements are the basic `LOOP, FOR LOOP and WHILE LOOP`
  - The `EXIT` statement transfers control to the end of a loop
  - The `CONTINUE` statement exits the current iteration of a loop and transfers control to the next iteration
  - both `EXIT` and `CONTINUE` have an optional `WHEN` clause, where you can specify a condition
- **sequential control statements**, which are not crucial to plsql programming. The sequential control statements are `GOTO`, which goes to a specified statement, and `NULL`, which does nothing

Conditional selection: IF

- the `IF` statement either runs or skips a sequence of one or more statement, depending on a condition
- the `IF` statement has these forms: `IF THEN`, `IF THEN ELSE`, `IF THEN ELSIF`
- `IF THEN` statement:
  - the `IF THEN` statement either runs or skips a sequence of one or more statements, depending on a condition
  - syntax:

```
IF condition THEN
    statements
END IF;
```

  - if the condition is true, the statements run; otherwise the `IF` statement does nothing
  - > Tip: avoid clumsy `IF` statements such as: `IF new_balance < minimum_balance THEN overdrawn := TRUE; ELSE overdrawn := FALSE; END IF;` instead, assign the value of the BOOLEAN expression directly to a BOOLEAN variable: `overdrawn := new_balance < minimum_balance`
  - > a `BOOLEAN` variable is either `TRUE`, `FALSE` or `NULL`, do not write: `IF overdrawn = TRUE THEN ... END IF;` instead write `IF overdrawn THEN ... END IF;`

- `IF THEN ELSE` statement:
  - syntax:

```
IF condition THEN
    statements
ELSE
    else_statements
END IF;
```

  - if the value of condition is true, the statements run; otherwise, the else_statements run
  - `IF` statements can be nested
- `IF THEN ELSIF` statement:
  - syntax:

```
IF condition1 THEN
    statements1
ELSIF condition2 THEN
    statements2
[ ELSIF condition3 THEN
    statements3
]...
[ ELSE
    else_statements
]
END IF;
```

  - the `IF THEN ELSIF` statement evaluates the boolean_expression conditions starting from condition1, then condition2 and so on until one is found to be true and runs the statements associated with it. Remaining conditions are not evaluated and the statements associated with them do not run
  - if no condition is true, the else_statements run, if they exist; otherwise, the `IF THEN ELSIF` statement does nothing
  - a single `IF THEN ELSIF` statement is easier to understand than a logically equivalent nested `IF THEN ELSE` statement
  - an `IF THEN ELSIF` statement with many `ELSIF` clauses to compare a single value to many possible values can be constructed more clearer with a simple `CASE` statement, they are logically equivalent in this case

Conditional selection: CASE

- the simple `CASE` statement:

- syntax:

```
CASE selector
    WHEN selector_value_1 THEN statements_1
    WHEN selector_value_2 THEN statements_2
    ...
    WHEN selector_value_n THEN statements_n
    [ ELSE
    else_statements ]
END CASE;
```

- the selector is an expression (typically a single variable)
- each selector_value can be either a literal or an expression of any plsql type except `BLOB, BFILE` or a user-defined type
- the simple `CASE` statement runs the first statements for which selector_value equals selector. Remaining conditions are not evaluated (The selector_values are evaluated sequentially. If the value of a selector_value equals the value of selector, then the statement associated with that selector_value runs, and the CASE statement ends. Subsequent selector_values are not evaluated.)
- if no selector_value equals selector, the `CASE` statement runs else_statements if they exist and raises the predefined exception `CASE_NOT_FOUND` otherwise (Without the `ELSE` clause, if no selector_value has the same value as selector, the system raises the predefined exception `CASE_NOT_FOUND`)
- > Note: if the selector in a simple `CASE` statement has the value NULL, it cannot be matched by `WHEN NULL`, instead use a searahed `CASE` statement with `WHEN condition IS NULL`

- Searched `CASE` statement
  - syntax:

```
CASE
    WHEN condition_1 THEN statements_1
    WHEN condition_2 THEN statements_2
    ...
    WHEN condition_n THEN statements_n
    [ ELSE
    else_statements ]
END CASE;
```

  - the searched `CASE` statement runs the first statements for which condition is true. Remaining conditions are not evaluated

    - if no condition is true, the `CASE` statement runs else_statements if they exist and raises the predefined exception `CASE_NOT_FOUND` otherwise
    - the else clause can technically be replaced by an `EXCEPTION` part if you omit it: `BEGIN CASE ... END CASE; EXCEPTION WHEN CASE_NOT_FOUND THEN ...`

☑Construct and identify loop statements

Basic loop: 4.2 - 4.2.1 For loop: 4.2.6 While loop: 4.2.7

- Loop statements run the same statements with a series of different values. The loop statements are:
    - basic `LOOP`
    - `FOR LOOP`
    - cursor `FOR LOOP`
    - `WHILE LOOP`
- The statements that exit a loop are:
    - `EXIT`
    - `EXIT WHEN`
- The statements that exit the current iteration of a loop are:
    - `CONTINUE`
    - `CONTINUE WHEN`
- `EXIT, EXIT WHEN, CONTINUE, and CONTINUE WHEN` can appear anywhere inside a loop, but not outside a loop (oracle recommends using these statements instead of the GOTO statement, which can exit a loop or the current iteration of a loop by transferring control to a statement outside the loop)
- a raised exception also exits a loop
- `LOOP` statements can be labeled, and `LOOP` statements can be nested. Labels are recommended for nested loops to improve readability. You must ensure that the label in the `END LOOP` statement matches the label at the beginning of the same loop statement (the compiler does not check)

Basic LOOP

- syntax: `LOOP statements END LOOP [label];`
- with each iteration of the basic `LOOP` statement, its statements run and control returns to the top of the loop
- the `LOOP` statement ends when a statement inside the loop transfers control outside the loop or raises an exception
- to prevent an infinite loop, atleast one statement must transfer control outside the loop (statements that can transfer control outside the loop are: `CONTINUE, EXIT, GOTO, RAISE`)
- can have a label that identifies the loop statement, `CONTINUE, EXIT, and GOTO` statements can reference this label

FOR LOOP

- syntax: `FOR index IN [REVERSE] lower_bound .. upper_bound LOOP statement END LOOP [label];`
    - the `FOR LOOP` statement runs one or more tatements while the loop index is in a specified range
    - without `REVERSE`, the value of index starts at lower_bound and increases by one with each iteration of the loop until it reaches upper_bound. If lower_bound is greater than upper_bound, then the statements never run
    - with `REVERSE`, the value of index starts at upper_bound and decreases by one with each iteration of the loop until it reaches lower_bound. If upper_bound is less than lower_bound, the the statements never run
    - an `EXIT, EXIT WHEN, CONTINUE, or CONTINUE WHEN` in the statements can cause the loop or the current iteration of the loop to end early
    - if lower_bound = upper_bound, then the loop executes exactly once, with or without `REVERSE`
- FOR LOOP index
    - the index of a `FOR LOOP` statement is implicitly declared as a variable of type `PLS_INTEGER` that is local to the loop
    - the statements in the loop can read the value of the index, but cannot change it. Statements outside the loop cannot reference the index
    - after the `FOR LOOP` statement runs, the index is undefined (a loop index is sometimes called a loop counter)
    - if the index of a `FOR LOOP` statement has the same name as a variable declared in an enclosing block, the local implicit declaration hides the other declaration. To use the other declaration, qualify it with a block label. You can also do this for indexes of nested `FOR LOOP` statements that have the same name
- Lower bound and upper bound
    - the lower and upper bounds of a `FOR LOOP` statement can be either numeric literals, numeric variables, or numeric expressions
    - if a bound does not have a numeric value, then plsql raises the predefined exception `VALUE_ERROR`
    - plsql evaluates lower_bound and upper_bound once, when the `FOR LOOP` statement is entered, and stores them as temporary `PLS_INTEGER` values, rounding them to the nearest integer if necessary
    - if lower_bound equals upper_bound, the statements run only once
- EXIT WHEN or CONTINUE WHEN statement in FOR LOOP statement
    - suppose that you must exit a `FOR LOOP` statement immediately if a certain condition arises. You can put the condition in an `EXIT WHEN` statement inside the `FOR LOOP` statement
    - suppose that the `FOR LOOP` statement that you must exit early is nested inside another `FOR LOOP` statement. If, when you exit the inner loop early, you also want to exit the outer loop, then label the outer loop and specify its name in the `EXIT WHEN` statement
    - if you want to exit the inner loop early but complete the current iteration of the outer loop, then label the outer loop and specify its name in the `CONTINUE WHEN` statement

WHILE LOOP statement

- syntax:

```
[label] WHILE condition LOOP
    statements
END LOOP [label];
```

- the `WHILE LOOP` statement runs one or more statements while a condition is true. If the condition is true, the statements run and control returns to the top of the loop, where condition is evaluated again. If condition is not true, control transfers to the statement after the `WHILE LOOP` statement
- to prevent an infinite loop, a statement inside the loop must make the condition false or null
- an `EXIT, EXIT WHEN, CONTINUE or CONTINUE WHEN` in the statements can cause the loop or the current iteration of the loop to end early
- some languages have a `LOOP UNTIL` or `REPEAT UNTIL` structure, which tests a condition at the bottom of the loop instead of at the top, so that the statements run atleast once. To simulate this structure in plsql, use a basic `LOOP` statement with an `EXIT WHEN` statement:

```
LOOP
  statements
  EXIT WHEN condition;
END LOOP;
```

☑Use EXIT and CONTINUE statements inside loops

EXIT: 4.2.2 - 4.2.3 CONTINUE: 4.2.4 - 4.2.5 GOTO: 4.3 - 4.3.1 NULL: 4.3.2

- unlike the `IF` and `LOOP` statement, the sequential control statements `GOTO` and `NULL` are not crucial to plsql programming
- the `GOTO` statement, which goes to a specified statement, is seldom needed. Occasionally, it simplifies logic enough to warrant its use
- the `NULL` statement, whihc does nothing, can improve readability by making the meaning and action of conditional statements clear

EXIT statement

- the `EXIT` statement exits the current iteration of a loop, either conditionally or unconditionally and transfers control to the end of either the current loop or an enclosing labeled loop
- an `EXIT` statement must be inside a `LOOP` statement
- syntax: `EXIT [label] [WHEN boolean_expression] ;`
- without label, the `EXIT` statement transfers control to the end of the current loop
- with label, the `EXIT` statement transfers control to the end of the loop that label identifies
- the `EXIT WHEN` statement exits the current iteration of a loop when the condition in its `WHEN` clause is true, and transfers control to the end of either the current loop or an enclosing labeled loop
- each time control reaches the `EXIT WHEN` statement, the condition in its `WHEN` clause is evaluated. If the condition is not true, the `EXIT WHEN` statement does nothing. To prevent an infinite loop, a statement inside the loop must make the condition true

CONTINUE statement

- the `CONTINUE` statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the next iteration of either the current loop or an enclosing labeled loop
- if a `CONTINUE` statement exits a cursor `FOR` loop prematurely (e.g. to exit an inner loop and transfer control to the next iteration of an outer loop), the cursor closes (in this context, `CONTINUE` works like `GOTO`)
- a `CONTINUE` statement must be inside a `LOOP` statement
- a `CONTINUE` statement cannot cross a subprogram or method boundary
- syntax: `CONTINUE [label] [WHEN boolean_expression] ;`
- without label, the `CONTINUE` statement transfers control to the next iteration of the current loop. With label, the `CONTINUE` statement transfers control to the next iteration of the loop that label identifies
- without `WHEN` clause the statement exits the current iteration of the loop unconditionally, with `WHEN` clause the statement exits the current iteration of the loop if and only if the value of boolean_expression is true

GOTO statement

- the `GOTO` statement transfers control to a label unconditionally, syntax: `GOTO label;`
- the label must be unique in its scope and must precede an executable statement or a plsql block
- when run, the `GOTO` statement transfers control to the labeled statement or block
- use `GOTO` statements sparingly. Overusing them results in code that is hard to understand and maintain
- do not use a `GOTO` statement to transfer control from a deeply nested structure to an exception handler. Instead raise an exception
- `GOTO` statement can transfer control to an enclosing block from the current block
- Restrictions on `GOTO` statement
    - `GOTO` statement cannot transfer control into an `IF` statement, `CASE` statement, `LOOP` statement, or sub-block
    - cannot transfer control from one `IF` statement clause to another, or from one `CASE` statement `WHEN` clause to another
    - cannot transfer control out of a subprogram
    - cannot transfer control into an exception handler
    - cannot transfer control from an exception handler back into the current block (but can transfer control from an exception handler into an enclosing block)

NULL statement

- the `NULL` statement only passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation)
- some uses for the NULL statement are:
    - to provide a target for a `GOTO` statement
    - to improve readability by making the meaning and action of conditional statements clear
    - to create placeholders and stub subprograms
    - to show that you are aware of a possibility, but that no action is necessary
- using the `NULL` statement might raise an unreachable code warning if warnings are enabled