Declarations

```
DECLARE
    fam_birthdate   DATE;
    fam_size        NUMBER(2) NOT NULL := 10;
    fam_location    VARCHAR2(13) := 'Florida';
    fam_population  INTEGER;
    fam_name        VARCHAR2(20) DEFAULT 'Roberts';
    fam_party_size  CONSTANT PLS_INTEGER := 20;
```

Using declared variables

```
DECLARE -- declarations
    v_myname    VARCHAR2(20);
BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myname);
    v_myname := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myname);
END;
```

Data types

Datatype	Explanation
`CHAR`	fixed-length chardata, default length 1
`VARCHAR2(max_length)`	variable-length chardata
`LONG`	chardata longer than `VARCHAR2`
`LONG RAW`	raw binary data
`NUMBER(p,s)`	number with precision p and scale s
`BINARY_INTEGER`	signed integer
`PLS_INTEGER`	faster than `NUMBER`
`BINARY_FLOAT` `BINARY_DOUBLE`	floating point numbers
`DATE`	stores a date
`TIMESTAMP`	extends `DATE`, holds up to fractions of seconds
`TIMESTAMP WITH TIME ZONE`	extends `TIMESTAMP`, includes timezone
`TIMESTAMP WITH LOCAL TIME ZONE`	extends `TIMESTAMP WITH TIME ZONE`, normalizes to database time zone
`INTERVAL YEAR TO MONTH`	stores intervals of years and months
`INTERVAL DAY TO SECOND`	stores intervals of days up to seconds
`BOOLEAN`	tri-state value: `TRUE`, `FALSE` or `NULL`

Useful keywords

Keyword	Explanation

`INTEGER`	alias for `NUMBER(38,0)`
`SYSDATE`	current date
`NOT NULL`	declares a variable that cannot be empty, needs to be initialized
`CONSTANT`	declares an unchgangeable variable, needs to be initialized?
`%TYPE`	can be used to dynamicaly get the datatype of a column: `table.col%TYPE`

Character functions

```
ASCII LENGTH RPAD
CHR LOWER RTRIM
CONCAT LPAD SUBSTR
INITCAP LTRIM TRIM
INSTR REPLACE UPPER
```

Number functions

```
ABS EXP ROUND
ACOS LN SIGN
ASIN LOG SIN
ATAN MOD TAN
COS POWER TRUNC
```

Date functions

```
ADD_MONTHS MONTHS_BETWEEN
CURRENT_DATE ROUND
CURRENT_TIMESTAMP SYSDATE
LAST_DAY TRUNC
```

Implicit conversions

(Implicit means no explicit conversion needed)

from/to	`DATE`	`LONG`	`NUMBER`	`PLS_INTEGER`	`VARCHAR`
`DATE`		YUP			YUP
`LONG`					YUP
`NUMBER`		YUP		YUP	YUP
`PLS_INTEGER`		YUP	YUP		YUP
`VARCHAR2`	YUP	YUP	YUP	YUP	

Some drawbacks

- may be slower
- making assumptions about future PL/SQL standards
- depending on environment (for example date formats)
- harder to read

Explicit conversions

- `TO_NUMBER()`
- `ROWIDTONCHAR()`

- `TO_CHAR()`
- `HEXTORAW()`
- `TO_CLOB()`
- `RAWTOHEX()`
- `CHARTOROWID()`
- `RAWTONHEX()`
- `ROWIDTOCHAR()`
- `TO_DATE()`

Order of operations

Operator	Operation
`**`	Exponentiation
`+, -`	Identity, negation
`*, /`	Multiplication, division
`+, -, ||`	Addition, subtraction, concatenation
`=, <, >, <=, >=,` `<>, !=, ~=, ^=,` `IS NULL, LIKE, BETWEEN, IN`	Comparison
`NOT`	Logical negation
`AND`	Conjunction
`OR`	Inclusion

Nested code blocks

In PL/SQL, one is allowed to nest code blocks, a variables' scope is this block and all nested code blocks. When a variable name is used, the one with the smallest scope is used.

To access variables with the same name, but that are not visible due to scope, one can use the following code:

```
<<outer>>
DECLARE -- outer block
    v_myname    VARCHAR2(20);
BEGIN
    v_myname := 'John';
    DECLARE -- inner block
        v_myname    VARCHAR2(20);
    BEGIN
        v_myname := 'Will';
        DBMS_OUTPUT.PUT_LINE('My name is: '||v_myname); -- Will
        DBMS_OUTPUT.PUT_LINE('My name is: '||outer.v_myname); -- John
    END;
END;
```

SQL statements

To retrieve data from a table, one can use a `SELECT` statement to put values into already declared variables. The query must return exactly one rown for this to work.

```
SELECT col1,col2 INTO v_col1, v_col2 FROM table (WHERE ...);
SELECT sum(col1) INTO v_sum_col1 FROM table (WHERE ...);
```

One can also use other SQL statements:

```
DELETE FROM table (WHERE ...);
INSERT INTO table (col1, col2, col3) VALUES ('param', 'values', 99)
UPDATE table SET col = newValue (WHERE ...)
```

WARNING: column names have smaller scope than variable names ergo `WHERE customer_id = customer_id` will not work (returns all rows).

Cursors

Implicit Cursors

Cursors are objects that carry the SQL statement. When an SQL statement is executed, one can use the implicit cursor "SQL" to gain information about that statement.

Explicit cursors

In the `DECLARE` block, one can declare a cursor and use it in a loop. Here is a simple example:

```
DECLARE
    CURSOR cursorName IS
        SELECT col1, col2 FROM table (WHERE ...);
    v_col1  table.col1%TYPE;
    v_col2  table.col2%TYPE;
BEGIN
    OPEN cursorName; -- executes the query
    LOOP
        FETCH cursorName INTO v_col1, v_col2; -- fetches a row
        EXIT WHEN cursorName%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_col1||' '||v_col2);
    END LOOP;
    CLOSE cursorName; -- required, clears memory
```

Variables can be defined to be of type `cursorName%ROWTYPE`, which is a rowobject. Fields in rows can be accessed through dotnotation. The above now becomes:

```
DECLARE
    CURSOR cursorName IS
        SELECT col1, col2 FROM table (WHERE ...);
    v_row   cursorName%ROWTYPE;
BEGIN
    OPEN cursorName; -- executes the query
    LOOP
        FETCH cursorName INTO v_row; -- fetches a row
        EXIT WHEN cursorName%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_row.col1||' '||v_row.col2);
    END LOOP;
    CLOSE cursorName; -- required, clears memory
```

Cursors can be opened while another cursor is open.

Cursor attributes

- `%FOUND:` Boolean value that is TRUE if the query returned at least one row.
- `%NOTFOUND:` Boolean value that is TRUE if the query returned no rows or there are no more rows to fetch.
- `%ROWCOUNT:` An integer containing the number of rows affected or the amount of rows already fetched.
- `%ISOPEN:` Evaluates whether the cursor is open.

WARNING: A cursor attribute cannot be used inside SQL statements.

Cursor FOR loops

One can shorten code a lot by using the cursor `FOR` loop. The following 2 statements are logically identical.

FOR	%ROWTYPE

`DECLARE`
`  CURSOR emp_cursor IS`
`  SELECT employee_id, last_name`
`  FROM employees`
`  WHERE department_id = 50;`
`BEGIN`
`  FOR v_emp_record IN emp_cursor`
`  LOOP`
`  DBMS_OUTPUT.PUT_LINE(...);`
`  END LOOP;`
`END`
	`DECLARE`
	`  CURSOR emp_cursor IS`
	`  SELECT employee_id, last_name`
	`  FROM employees`
	`  WHERE department_id = 50;`
	`  v_emp_record emp_cursor%ROWTYPE;`
	`BEGIN`
	`  OPEN emp_cursor;`
	`  LOOP`
	`  FETCH emp_cursor INTO v_emp_record;`
	`  EXIT WHEN emp_cursor%NOTFOUND;`
	`  DBMS_OUTPUT.PUT_LINE(...);`
	`  END LOOP;`
	`  CLOSE emp_cursor;`
	`END;`

One does not need to declare a cursor, but can insert the SQL statement inside a `FOR` declaration:

```
FOR v_emp_record IN SELECT employee_id, last_name FROM employees WHERE department_id = 50
LOOP
    DBMS_OUTPUT.PUT_LINE(...);
END LOOP;
```

Cursor parameters

A cursor can be defined with a parameter, which can be called when opening the cursor. This is very useful, as the statement in the cursor gets executed each time it is opened.

```
DECLARE
    CURSOR cursorName (p_param NUMBER) IS
        SELECT * FROM TABLE WHERE id = p_param;
    ...
BEGIN
    OPEN cursorName(5);
    ...
```

Cursor locking

A cursor can prevent the rows it is acting upon from being changed.

```
CURSOR cursorName IS
SELECT ... FROM ...
FOR UPDATE (OF column) (NOWAIT | WAIT n);
```

Specific columns can be locked by using the `OF` parameter.

If the rows are locked by another session, then the parameters `NOWAIT` and `WAIT n` come into play

- `NOWAIT` returns an error immediately
- `WAIT n` waits for n seconds before trying again, if still locked, returns an error

Update by cursor

```
UPDATE table
```

```
    SET column = ...
    WHERE CURRENT OF cursorName; -- takes the
```

Transaction control

A number of commands can be used to control transactions:

- `COMMIT` is used to finalize the previous statement and permanently write them to the database.
- `ROLLBACK` rolls the state of the db back to the last `COMMIT`.
- `SAVEPOINT` is used so one can `ROLLBACK` in steps.

Control structures

If/else structure

```
IF condition THEN
    statements; -- only executes these if condition is TRUE, not if FALSE or NULL
(ELSIF condition THEN
    statements;)
(ELSE
    statements;)
END IF;
```

WARNING: `NULL` is an unknown value, not an empty one. `NULL = NULL` results in `NULL`, as both are unknown.

Case structure

```
CASE v_var -- shorter notation for IF/ELSIF/ELSE with the same variable
    WHEN 'A' THEN
        statements; -- same as IF v_var = 'A'
    WHEN 'B' THEN
        statements;
    ELSE
        statements;
END CASE;
```

A `CASE` statement can also be used in an assignment:

```
v_var :=
    CASE v_other_var
        WHEN 1 THEN 'One'
        WHEN 2 THEN 'Two'
        ELSE 'Many'
    END; -- note the lack of "CASE" here
```

Another use is the searching expression, which is a simplified form of `IF/ELSIF/ELSE`.

```
v_var :=
    CASE -- no selector here
        WHEN condition1 THEN 'One'
        WHEN condition2 THEN 'Two'
        ELSE 'Many'
    END; -- note the lack of "CASE" here
```

Loop structure

A simple loop looks like this:

```
LOOP
```

```
    statements;
END LOOP;
```

In the statements must be at least one `EXIT` statement, which can be formed like this:

```
EXIT; -- to use inside an IF statement in the loop
EXIT WHEN condition; -- simplifies the IF
```

`WHILE` and `FOR` loops are also available:

```
WHILE condition LOOP
    statements;
END LOOP;
```

```
FOR counter IN (REVERSE) lower..upper LOOP -- counter is defined in the loop as an INTEGER
                                           -- lower and upper are included in the loop 1..3 loops 3 times
    statements;
END LOOP;
```

Nested loops can be used, and the `EXIT` keyword exits the smallest loop. To exit multiple loops at the same time, labels can be given, identical to the scope labels.

```
DECLARE
    declarations;
BEGIN
    <<outer_loop>>
    LOOP -- outer loop
        <<inner_loop>>
        LOOP -- inner loop
            EXIT outer_loop (WHEN ...) -- Exits both loops
            EXIT WHEN v_inner_done;
        END LOOP;
        EXIT WHEN v_outer_done;
    END LOOP;
END;
```

User-defined records

`%ROWTYPE` can be used on cursors, tables and other rowtypes.

Custom records can be created using the following syntax in the `DECLARE` block:

```
TYPE custom_type IS RECORD
    (col1       table.col1%TYPE,
    col2        VARCHAR2(60));
```

Which can then be used like this:

```
DECLARE
    v_cust_rec  custom_type;
```

These can also be nested (A custom type can contain fields of *another* custom type).

Tables of records

```
DECLARE
    TYPE t_names IS TABLE OF VARCHAR2(50)
                    INDEX BY BINARY_INTEGER;
```

Accessing by primary key:

```
v_a_table(variable) := something;
```

A few methods, used by dot-notation:

- `EXISTS` function to check whether a row exists with this primary key
- `COUNT` property of a table, returns the length
- `FIRST` property of a table, returns the first primary key
- `LAST` property of a table, returns the last primary key
- `PRIOR`
- `NEXT`
- `DELETE`
- `TRIM`

The index can only be one field, the data can be a composite data type, like a rowtype.

Exceptions

Exceptions can be caught by using the following syntax:

```
DECLARE
    v_country_name wf_countries.country_name%TYPE := 'Korea, South';
    v_elevation wf_countries.highest_elevation%TYPE;
BEGIN
    SELECT highest_elevation INTO v_elevation
        FROM wf_countries WHERE country_name = v_country_name;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('Country name, '|| v_country_name ||', cannot be found. Re-enter the country name using the correct
    WHEN ... (OR ...) THEN
        ...
END;
```

List of exceptions:

- `NO_DATA_FOUND`
- `TOO_MANY_ROWS`
- `OTHERS` matches all exceptions, can be used as catch-all-exceptions
- `INVALID_CURSOR`
- `ZERO_DIVIDE`
- `DUP_VAL_ON_INDEX`

Other exceptions (without name) can be caught when defined:

```
DECLARE
    e_insert EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_insert, -01400);
BEGIN
    ...
EXCEPTION
    WHEN e_insert
        THEN
            ...
END;
```

`SQLCODE NUMBER` and `SQLERRM VARCHAR2(255)` can be used in an exceptionblock to get more info.

One can raise an existing exception using the keyword `RAISE`, or a new one without defining one with `RAISE_APPLICATION_ERROR (errno, 'message');`. Raising a customdefined exception is possible like so:

```
DEFINE
    e_invalid_nr    EXCEPTION;
BEGIN
```

```
    RAISE e_invalid_nr;
EXCEPTION
    WHEN e_invalid_nr THEN
        ...
END;
```

Scope of exceptions

An exception will halt execution of all code until it is handled. Use nested blocks to handle exceptions when code needs to be executed after an exception is handled. In this example when there is more than one `employee_ID` 999, messages 2 and 3 will be shown; when there is no `employee_id` 999, then message 4 will be shown:

```
DECLARE
    v_last_name employees.last_name%TYPE;
BEGIN
    BEGIN
        SELECT last_name INTO v_last_name
            FROM employees WHERE employee_id = 999;
        DBMS_OUTPUT.PUT_LINE('Message 1');
    EXCEPTION
        WHEN TOO_MANY_ROWS THEN
            DBMS_OUTPUT.PUT_LINE('Message 2');
    END;
    DBMS_OUTPUT.PUT_LINE('Message 3');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Message 4');
END;
```

IMPORTANT: When declaring exceptions, always declare them in the outermost block, as you otherwise can no longer catch the exception outside the block with its declaration.

Procedures

```
CREATE (OR REPLACE) PROCEDURE proc_name
    (param1 (mode) DATATYPE (DEFAULT value),
    ...)
IS
    declarations;
BEGIN
    statements;
(EXCEPTION
    WHEN ... THEN ...;)
END;
```

A procedure can be ended prematurely by calling `RETURN;`

Us the optional part `OR REPLACE` to overwrite a previous `PROCEDURE` with the same name.

IMPORTANT: As with everything else in PL/SQL, you cannot invoke a procedure in an SQL statement, put the result in a variable first if you need to.

invoking a procedure is simply calling `proc(params);`.

Subprocedure

One can declare a subprocedure in the declarations of a main procedure by using this syntax:

```
PROCEDURE subproc (...) IS
BEGIN
    statements
END subproc;
```

In/out/in out parameters

- `IN`-parameters are normal parameters, cannot be changed in the procedure.
- `OUT`-parameters are empty variables in the calling environment, which are filled in the procedure.
- `IN OUT`-parameters are both at the same time, they send a value to the procedure and get modified.

Passing parameters

Parameters can be sent by order (same as procedure definition), by name, using `proc(param2=>v_p2, param1=>v_p1);`, or both. When using both (**strongly** discouraged), positional parameters come first.

Dropping procedures

```
DROP PROCEDURE proc_name;
```

Functions

Similar to a procedure, only contains `IN`-parameters and returns one value.

```
CREATE OR REPLACE FUNCTION func_name
    (param1 DATATYPE (DEFAULT value),
    ...)
RETURN DATATYPE
IS
    declarations;
BEGIN
    statements;
    RETURN value;
(EXCEPTION
    WHEN ... THEN ...;)
END;
```

IMPORTANT: These can be used in SQL statements, and there is only positional notation for parameters.

Dropping functions

```
DROP FUNCTION func_name;
```

Privileges

```
GRANT privileges (columns) ON object TO user|role|PUBLIC
```

Privileges are one of the following:

- `ALTER`
- `DELETE`
- `EXECUTE`
- `INDEX`
- `INSERT`
- `REFERENCES` check for existence
- `SELECT`
- `UPDATE`

and similarly, `REVOKE` with identical syntax.

When using a multi-user environment, the references are with the definer, so that if a user calls a function from another user, he or she is using

the tables of the person that defined that function.

```
AUTHID CURRENT_USER IS BEGIN
    statements; -- these get executed in the invoker's environment
END;
```

Packages

A package contains a specification, and a body. The calling environment can only see the specification.

```
CREATE (OR REPLACE) PACKAGE pack_name
AS
    v_var       VARCHAR2(60);
    PROCEDURE proc_name (p_param1, ...);
    CURSOR c_curs IS SELECT * FROM table;
END

CREATE (OR REPLACE) PACKAGE BODY pack_name
AS
    PROCEDURE proc_name
        (p_param1, ...) IS
    BEGIN
        statements;
    END;
END;
```

To get information about a package: `DESCRIBE pack_name`. Packages get stored in the table `USER_SOURCE`:

```
SELECT text
    FROM user_source
    WHERE name = 'PACK_NAME' AND type = 'PACKAGE' -- use 'PACKAGE BODY' to see the body
    ORDER BY line;
```

Subprograms can be overloaded when the parameters are entirely different (amount of parameters or completely different types (CHAR and VARCHAR2 are too similar)).

private subprograms and variables must be declared before they are used. Their body can be somewhere else if the declaration is available.

Visibility

Everything inside package specs is visible to anyone with permission. Everything inside package body is only visible to package body.

To call anything inside a package from the outside, use `pack_name.name`.

Dropping packages

One can choose to drop both the specs and the body by using `DROP PACKAGE pack_name`, or drop only the body by using `DROP PACKAGE BODY pack_name`.

Bodiless packages

Packages with only initialized variables can exist without a body.

Performance

Performance can be improved by using the `NOCOPY` keyword when defining `OUT` mode parameters in a procedure. This ensures the variable is passed by reference and not by value, thus not copying the data.

```
PROCEDURE proc_name (p_param1 OUT NOCOPY DATATYPE, ...)
```

The keyword `DETERMINISTIC` can be used when a function always returns the same if the same input is used.

```
    RETURN DATATYPE DETERMINISTIC
```