DB1 Syntax Details

Felix Pojtinger

June 22, 2021

Contents

DB1 Syntax Details

"so basically i am monkey" - monke, *monkeeee*

Acknowledgements

Most of the following is based on the Oracle Tutorial.

Reset Everything

Run the following to get the commands to drop all tables and their constraints:

```
begin
  for i in (select index_name from user_indexes where index_name not like '%_PK') loop
    execute immediate 'drop index ' || i.index_name;
  end loop;

  for i in (select trigger_name from user_triggers) loop
    execute immediate 'drop trigger ' || i.trigger_name;
  end loop;

  for i in (select view_name from user_views) loop
    execute immediate 'drop view ' || i.view_name;
  end loop;

  for i in (select table_name from user_tables) loop
    execute immediate 'drop table ' || i.table_name || ' cascade constraints';
  end loop;

  execute immediate 'purge recyclebin';
end;
```

Now copy & paste the output into SQL Developer's SQL worksheet and hit F5.

SQL

Operators

Operator	Description
=	Equality
!=,<>	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
IN	Equal to any value in a list of values
ANY/ SOME/ ALL	Compare a value to a list or subquery. It must be preceded by another operator such as =, >, <.j
NOT IN	Not equal to any value in a list of values
[NOT] BETWEEN n and m	Equivalent to [Not] >= n and <= y.
[NOT] EXISTS	Return true if subquery returns at least one row
IS [NOT] NULL	NULL test

Joins

- An **inner join** matches stuff in both tables:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_
```

- A **left (outer) join** matches everything in the left tables plus what matches in the right table:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_
```

- This **left (outer)** join matches everything that is in the left table and not in the right table:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_
```

- A **right (outer) join** matches everything in the right join plus what matches in the left table:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_
```

- This **right (outer) join** matches everything that is in the right table and not in the left table:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_
```

- A **full (outer) join** merges both tables:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_
```

- This **full (outer) join** merges both tables and removes those rows which are in both:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_
```

- In addition to the on keyword you can also use the using keyword if the PK and FK are the same:

```
select * from orders inner join order_items using(order_id)
```

- You can also use multiple on or using statements:

```
select * from orders inner join order_items using(order_id) inner join customers using(
```

- If you use the on keyword, use `and` for multiples!
- You can also create the Cartesian product:

```
select * from products cross join warehouse;
```

- It is also possible to do a self join:

```
select (w.first_name || ' ' || w.last_name) "Worker", (m.first_name || ' ' || m.last_na
```

- What is the difference between `join` and `union`? `join` merges horizontally (there are more columns than before, maybe also more rows), `union` merges vertically (there are more rows than before, but the column count stays the same).
- `union` is similar to `T1 | T2` in TypeScript; you can use order by and union to remove duplicates, but note that we have to use select two times:

```
select first_name, last_name, email, 'contact' as role from contacts union select first
```

- `union all` is similar to `T1 & T2` in TypeScript; it keeps duplicates:

```
select last_name from contacts union all select last_name from employees;
```

- Wish to find the difference between two tables? Use `intersect`:

```
select last_name from contacts intersect select last_name from employees;
```

- Wish to subtract one table from another table? Use `minus`:

```
select last_name from contacts minus select last_name from employees;
```

Aliases

- You can alias long column names with `select mylongname as name from contacts` or just `select mylongname name from contacts`. The `as` keyword is optional. Full-text column names are supported by enclosing in “”. as can also format strings: `select first_name || ' ' || last_name as "Name" from employees;` yields Alice, Bob and System.

- You can also create a table alias (using `from employees e`), but you CAN'T USE the `as` keyword.

Limits and Pagination

- The Oracle equivalent of filter is `fetch n next rows only`: `select * from products order by list_price desc fetch next 5 rows only;`.
- You may also use the `fetch next n percent rows only`:

```
select * from inventories order by quantity desc fetch next 10 percent rows only;
```

- Filtering by for example a quantity, and you only want the first 10 "condition matches"? Use `fetch n next rows with ties`:

```
select * from inventories order by quantity desc fetch next 5 rows with ties;
```

- Need Pagination? Use offset:

```
select * from products order by standard_cost desc offset 10 rows fetch next 10 rows or
```

Dates and Intervals

- Want to extract a year from a date? Use `extract`:

```
select * from orders where status = 'Shipped' and extract(year from order_date) = 2017
```

- Want to get the current date? Use `current_date`:

```
select current_date from dual;
```

- The `to_char` function can convert dates (and timestamps) to chars:

```
select to_char(sysdate, 'YYYY-MM-DD') from dual;
```

- The `to_date` function can convert chars to dates:

```
select to_date('2021-01-12', 'YYYY-MM-DD') from dual;
```

- Alternatively, the date literal uses the `YYYY-MM-DD` format and does not require format specs:

```
select date '1969-04-20' from dual;
```

- You can get the current date with `sysdate`:

```
select localtimestamp from dual;
```

- You can get the current date & time with `datelocaltimestamp`:

```
select localtimestamp from dual;
```

- The current time zone is available with `sessiontimezone`:

```
select sessiontimezone from dual (yields Europe/Berlin);
```

- The `timestamp` literal uses the `YYYY-MM-DD HH24:MI:SS.FF` format:

  ```
  select timestamp '1969-04-20 00:00:00.00' from dual;
  ```

- You may also append the timezone (But keep in mind that timestamp with time zone is the column type in this case):

  ```
  select timestamp '1969-04-20 00:00:00.00 Europe/Berlin' from dual;
  ```

- The `interval` literal can be used to create intervals:

  ```
  select interval '9' day from dual, select interval '9' month from dual, select interval
  ```

- Using the `months_between` function, the count of months between two dates can be computed.

Expressions

- Only single quotes are supported.
- Comparisons are done with `=`, NOT `==`.
- It also supports full expression evaluation:

```
select product_name as "Product Name", list_price - standard_cost as "Gross Profit" from pro
```

- You can use `()` in `where` clauses to prioritize:

  ```
  select * from orders where (
  status = 'Canceled' or status = 'Pending' ) and customer_id = 44
  order by order_date;
  ```

- The `in` keyword is a useful tool for sub collections and subqueries:
  - `select * from orders where salesman_id in (54, 55, 56) order by order_id;`
  - `select * from orders where salesman_id not in (54, 55, 56) order by order_id;` (you can use not)
  - `select * from employees where employee_id in ( select distinct salesman_id from orders where status = 'Canceled' ) order by first_name;` (you can of course also use not)

- `between` can also be used for dates:

  ```
  select * from orders where order_date between date '2016-12-01' and date '2016-12-31'
  ```

- `... like '%Asus%'` (note the 's) is basically a full-text search.
- Some examples of `like` (you can use `not` for all of them):
  - `select * from contacts where last_name like 'St%'`
  - `select * from contacts where last_name like '%St'`
  - `select * from contacts where last_name like '%St%'`
  - `select * from contacts where last_name like 'Po_tinger'` (_ matches any one character)
  - `select * from contacts where lower(last_name) like 'st%'`

  - `select * from contacts where upper(last_name) like 'st%'`
  - `select * from discounts where discount_message like '%%%'` (returns everything)
  - `select * from discounts where discount_message like '%%%' escape '!'` (returns everything that includes the string '%')
- You can compare against null with `is null` (`= NULL` does not work). You can negate with `not`.

Grouping and Ordering

- You can use functions like `upper` and dates when ordering.
- The `group by` keyword can be used to find unique data:

```
select status from orders group by status;
```

- By combining `group by` with `count` you can count the amount of unique data:

```
select status, count (*) from orders group by status;
```

- `group by` can also be used with the `where` keyword:

```
select name, count(*) as "Shipped Orders" from orders inner join customers using(custom
```

- `where` can NOT APPEAR AFTER `group by`; use the having keyword instead.
- The `having` keyword enables you to filter like with `where`, but after the `group by` keyword like so:

```
select status from orders where extract(year from order_date) > '2016' group by status
```

- Multiple order by statements? First ordered by first statement, then "sub-ordered" by the second (last name the same -> now first name is evaluated).
- Want to have nulls first when ordering? Use nulls first or nulls last as the suffix.
- Removal of duplicates is done with `select distinct`. When multiple columns are being selected, use only one distinct keyword at the start. Multiple nulls are filtered (Null = Null).

Counting and Sums

- You can count the amount of rows with the `count()` function:

```
select count(*) from products
```

- The `sum` function can be used to calculate a total:

```
select sum(unit_price * quantity) from order_items;
```

- It can also be used to calculate a total per row (the group by `order_id` part is required; `group by order_value` does not work):

  ```
  select order_id, sum(unit_price * quantity) as order_value from order_items group by or
  ```

Inserting

- It is a good idea to always specify the columns when inserting:

  ```
  insert into discounts(discount_name, amount, start_date, expired_date) values ('Summer
  ```

- You can also "insert from select" using `insert into`:

  ```
  insert into sales(customer_id, product_id, order_date, total) select customer_id, produ
  ```

- It's even possible to "create a table from select" using `create table x as`, basically coping its schema (`where 1 = 0 skips` copying the rows):

  ```
  create table sales_2017 as select * from sales where 1 = 0;
  ```

- Using insert all, it is possible to insert multiple rows at once (note the lack of commas between the into keywords. Here, the subquery is ignored/a placeholder.):

  ```
  insert all into fruits (fruit_name, color) values ('Apple', 'Red') into fruits (fruit_n
  ```

- You can also use conditions based on the subquery (`insert first` is the equivalent of a switch case.):

  ```
  insert all when amount < 10000 then into small_orders when amount >= 10000 then into bi
  ```

Switches

- Using `case` it is possible to create if/else constructs:

  ```
  select product_name, list_price, case category_id when 1 then round(list_price * 0.05,
  ```

- `case` is also useful for conditional grouping:

  ```
  select * from locations order by country_id, case country_id when 'US' then state else
  ```

- `case` also evaluates to an expression, so you can use it for conditional updates:

  ```
  update products set list_price = case when list_price < 20 then 30 else 50 end where li
  ```

Helper Functions

- You can extract substrings with `substr`: `select substr('Alex', 1, 1) from dual;`
- Stuff like `select upper('uwu') from dual` can come in handy.
- Using `round` it is possible to round numbers (returns 5.23):

```
select round(5.234234234234, 2) from dual;
```

- You can use `replace` to replace strings:

```
update accounts set phone = replace(phone, '+1-', '');
```

- You can use the `floor`, `round` and `ceil` functions to get rounded values.

Auto-Generated Primary Keys

- `generated by default as identity` is quite useful for auto-incrementing columns such as PKs:

```
create table persons ( person_id number generated by default as identity, first_name va
```

- `generated always as identity` is the same but does not allow setting it manually.

Modifying Columns

- You can use `desc mytable` to show the schema for a table.
- `alter table` can be used to add columns using add:

```
alter table persons add birthdate date not null;
```

- You can also add multiples at once (note that there is no column keyword):

```
alter table persons add ( phone varchar2(20), email varchar2(100) )
```

- `modify` can change the column type (note that there is no column keyword):

```
alter table persons modify birthdate date null;
```

- `drop column` can be used to remove a column

```
alter table persons drop column birthdate;
```

- `rename column` can be used to rename a column:

```
alter table persons rename column first_name to forename;
```

- `rename to` can be used to rename a table:

```
alter table persons rename to people;
```

- `rename promotions to promotions_two` is an alternative syntax.
- You can use the default keyword to set a default value:

```
alter table accounts add status number(1,0) default 1 not null.
```

- A more efficient logical version of drop column is set unused column:

```
alter table suppliers set unused column fax;
```

- You can now drop it using:

```
alter table suppliers drop unused columns;
```

- If you want to physically drop a column, use `drop`:

```
alter table suppliers drop (email, phone);
```

Virtual Columns

- You can create virtual columns in regular tables without using views with `alter table x add ... as` (note the required `(` after the `as` keyword):

```
alter table parts add (capacity_description as ( case when capacity <= 8 then 'Small' w
```

- The size of a `varchar2` is adjustable afterwards (note that this checks if any current `varchar2`s are larger than the new size and fails if they are.):

```
alter table persons modify first_name varchar2(255);
```

Modifying Tables

- You can drop a table with `drop table`:

```
drop table people;
```

- Appending `purge` clears the recycle bin; appending `cascade constraints` drop all related constraints.
- You can clear a table using `truncate table`:

```
truncate table customers_copy;
```

- The same limitations as with drop table concerning constraints apply, so appending cascade (WITHOUT constraints) drops all related ones.
- You can clear the recycle bin with:

```
purge recyclebin;
```

Constraints

- It is possible to add constraints (any constraints, a primary key in this example) after creating a table with add constraint:

```
alter table purchase_orders add constraint purchase_orders_order_id_pk primary key(orde
```

- You may remove a constraint with `drop constraint`:

```
alter table purchase_orders drop constraint purchase_orders_order_id_pk;
```

- Instead of removing it, you can also use `disable constraint`:

```
alter table purchase_orders disable constraint purchase_orders_order_id_pk;
```

- And re-enable it with `enable constraint`:

```
alter table purchase_orders enable constraint purchase_orders_order_id_pk;
```

- You can also add foreign key constraints:

  ```
  alter table suppliers add constraint suppliers_supplier_groups_fk foreign key(group_id)
  ```

- Using a check constraint, arbitrary expressions can be evaluated:

  ```
  alter table parts add constraint check_buy_price_positive check(buy_price > 0);
  ```

- A unique constraint prevents unwanted duplicates:

  ```
  alter table clients add constraint unique_clients_phone unique(phone);
  ```

- With a not null constraint, fuzzy logic can be avoided; it is however best to define nullable fields at schema creation, as the syntax differs from the add constraint/drop constraint logic above:

  ```
  alter table clients modify ( 7 phone not null );
  ```

- You can remove them by modifying it to null explicitly:

  ```
  alter table clients modify ( phone null );
  ```

Types

- You can create a number within a range: `number(1,0)`.
- The `number` type is used for all types of numbers by specifying precision and scale: `number(6)` (or `number(6,0)`) is a signed integer fitting 6 digits, `number(6,2)` is a float with two digits precision. The DB doesn't just cut of numbers, it rounds them.
- The float type can be emulated by the number type, i.e. `float(2)` is equal to `number(38,2)`. The argument is in bits instead of digits though.
- The `lengthdb` function can be used to get the length of field in bytes.
- The char type has a fixed length: name `char(10)` or name `char(10 bytes)`, meaning that a char always takes up the amount of bytes set. `nchar` is the same but UTF-8 or UTF-16 any doesn't take bytes.
- The `varchar2` type also takes an argument for the length in bytes, which in ASCII corresponds to the amount of characters. `nvarchar2` is the same but UTF-8 or UTF-16 and doesn't take bytes.

Views

- You can create a view with `create view x as select ...`:

  ```
  create view employees_years_of_service as select employee_id, first_name || ' ' || last
  ```

- If used with `create or replace view`, upserts are possible.
- By appending `with read only`, you can prevent data modifications:

  ```
  create or replace view employees_years_of_service as select employee_id, first_name ||
  ```

- `drop view x` removes the view.

- Deletions and updates on views are usually fine, but inserts can often be not that useful due to fields being excluded from the view; see `instead of` triggers later on for a solution;
- Subqueries can be used in selects:

  ```
  select * from ( select * from products) where list_price < 100;
  ```

- They can also be used in updates:

  ```
  update ( select list_price from products ) set list_price = list_price * 1.5;
  ```

Indexes

- You can create an index with `create index`:

  ```
  create index members_last_name on members(last_name);
  ```

- You can also create an index spanning multiple columns:

  ```
  create index members_full_name on members(first_name, last_name);
  ```

- You can drop an index with `drop index`:

  ```
  drop index members_full_name;
  ```

PL/SQL

Block Structure

- Block structure:

  ```
  declare
  -- declarations
  begin
  -- your logic
  exception
  -- exception handling
  end;
  ```

- The most simple example is as follows:

  ```
  begin
     dbms_output.put_line('Hello World!');
  end;
  ```

- Use `put_line` from the `dmbs_output` package to print to stdout.
- You can use the declare section for variables:

  ```
  declare
      message varchar(255) := 'Hello, World!';
  begin
  ```

```
    dbms_output.put_line(message);
end;
```

- The `exception` block is used to handle exceptions, for example `zero_divide` for divisions by zero (`when others then` handles unexpected other exceptions):

```
declare
    result number;
begin
    result := 1/0;

    exception
        when zero_divide then
            dbms_output.put_line(sqlerrm);
        when others then
            dbms_output.put_line('An unexpected error occured: ' || sqlerrm);
end;
```

- You always have to specify an execution section; use `null` for a no-op:

```
declare
begin
    null;
end;
```

- You can use `--` for single line comments and `/*` for multi line comments.

Variables

- PL/SQL extends SQL by adding a boolean type (which can have the values true, false and null).
- Variables need not be given a value at declaration if they are nullable:

```
declare
    total_sales number(15,2);
    credit_limit number(10,0);
    contact_name varchar2(255);
begin
    null;
end;
```

- You can use `default` as an alternative to the `:=` operator when assigning variables in the declaration section. DO NOT use `=` when assignment, even re-assignment also uses `:=`.
- If a variable is defined as not null, it can't take a string of length 0:

```
declare
    shipping_status varchar2(25) not null := 'shipped';
```

```
begin
    shipping_status := ''; -- You need to specify any string != ''
end;
```

- Constants are created with the `constant` keyword and forbid reassignment:

```
declare
    price constant number := 10;
begin
    price := 20; -- Will throw an exception
end;
```

Fetching Data

- Use `select ... into` to fetch data into variables; `%TYPE` infers the type of a column:

```
declare
    customer_name customers.name%TYPE;
    customer_credit_limit customers.credit_limit%TYPE;
begin
    select
        name, credit_limit
    into
        customer_name, customer_credit_limit
    from customers where customer_id = 38;

    dbms_output.put_line(customer_name || ': ' || customer_credit_limit);
end;
```

Branches and Expressions

- `if ... then ... end if` can be used for branching:

```
declare
    sales number := 20000;
begin
    if sales > 10000 then
        dbms_output.put_line('Lots of sales!');
    end if;
end;
```

- Inline expressions are also supported:

```
large_sales := sales > 10000
```

- Booleans need not be compared with `my_bool = true`, a simple `if my_bool then` is fine.
- `elseif ... then` is NOT valid syntax; `elsif ... then` is valid syntax.

- Statements may also be nested:

```
declare
    sales number := 20000;
begin
    if sales > 10000 then
        if sales > 15000 then
            dbms_output.put_line('A new sales record!');
        else
            dbms_output.put_line('Lots of sales!');
        end if;
    end if;
end;
```

Switches

- You may use the `case` keyword for switch cases:

```
declare
    grade char(1);
    message varchar2(255);
begin
    grade := 'A';

    case grade
        when 'A' then
            message := 'Excellent';
        when 'B' then
            message := 'Great';
        when 'C' then
            message := 'Good';
        when 'D' then
            message := 'Fair';
        when 'F' then
            message := 'Poor';
        else
            raise case_not_found;
    end case;

    dbms_output.put_line(message);
end;
```

Labels and Goto

- A `label`/`goto` equivalent is also available:

```
begin
    goto do_work;
```

```
    goto goodbye;

    <<do_work>>
    dbms_output.put_line('mawahaha');

    <<goodbye>>
    dbms_output.put_line('Goodbye!');
end;
```

Loops

- The equivalent of the `while` loop is the `loop`. `exit`/`continue` prevents an infinite loop:

```
declare
    i number := 0;
begin
    loop
        i := i + 1;

        dbms_output.put_line('Iterator: ' || i);

        if i >= 10 then
            exit;
        end if;
    end loop;

    dbms_output.put_line('Done!');
end;
```

- For loops can be done using the `for i in 0..100 loop ... end loop` syntax:

```
begin
    for i in 0..100 loop
        dbms_output.put_line(i);
    end loop;
end;
```

- While loops work as you'd expect; but also require the `loop` keyword:

```
declare
    i number := 0;
begin
    while i <= 100 loop
        dbms_output.put_line(i);

        i := i + 1;
```

```
    end loop;
end;
```

Types and Objects

- You can also use `%ROWTYPE` to infer the type of a row and select an entire row at once:

```
declare
    customer customers%ROWTYPE;
begin
    select * into customer from customers where customer_id = 100;

    dbms_output.put_line(customer.name || '/' || customer.website);
end;
```

- It is also possible to use OOP-style object/row creation thanks to `%ROWTYPE`:

```
declare
    person persons%ROWTYPE;

begin
    person.person_id := 1;
    person.first_name := 'John';
    person.last_name := 'Doe';

    insert into persons values person;
end;
```

Exceptions

- You can create custom exceptions:

```
declare
    e_credit_too_high exception;
    pragma exception_init(e_credit_too_high, -20001);
begin
    if 10000 > 1000 then
        raise e_credit_too_high;
    end if;
end;
```

- If you want to raise a custom exception, use `raise_application_error`:

```
declare
    e_credit_too_high exception;
    pragma exception_init(e_credit_too_high, -20001);
begin
```

```
    raise_application_error(-20001, 'Credit is to high!');
end;
```

- Using `sqlcode` and `sqlerrm` you can get the last exception's code/error message.

Cursors

- Using cursors, you can procedurally process data:

```
declare
    cursor sales_cursor is select * from sales;
    sales_record sales_cursor%ROWTYPE;
begin
    update customers set credit_limit = 0;

    open sales_cursor;

    loop
        fetch sales_cursor into sales_record;
        exit when sales_cursor%NOTFOUND;

        update
            customers
        set
            credit_limit = extract(year from sysdate)
        where
            customer_id = sales_record.customer_id;
    end loop;

    close sales_cursor;
end;
```

- Complex exit logic can be avoided using the `for ... loop`:

```
declare
    cursor product_cursor is select * from products;
begin
    for product_record in product_cursor loop
        dbms_output.put_line(product_record.product_name || ': $' || product_record.lis
    end loop;
end;
```

- Cursors can also have parameters:

```
declare
    product_record products%rowtype;
    cursor
        product_cursor (
```

```
            low_price number := 0,
            high_price number := 100
        )
    is
        select * from products where list_price between low_price and high_price;
begin
    open product_cursor(50, 100);

    loop
        fetch product_cursor into product_record;
        exit when product_cursor%notfound;

        dbms_output.put_line(product_record.product_name || ': $' || product_record.lis
    end loop;

    close product_cursor;
end;
```

Locks

- The DB can also lock fields for safe multiple access:

```
declare
    cursor customers_cursor is select * from customers for update of credit_limit;
begin
    for customer_record in customers_cursor loop
        update customers set credit_limit = 0 where customer_id = customer_record.custo
    end loop;
end;
```

Procedures

- You can create procedures, which are comparable to functions:

```
create or replace procedure
    print_contact(customer_id_arg number)
is
    contact_record contacts%rowtype;
begin
    select * into contact_record from contacts where customer_id = customer_id_arg;

    dbms_output.put_line(contact_record.first_name || ' ' || contact_record.last_name);
end;
```

- These procedures can then be executed:

```
begin
    print_contact(50);
```

```
end;
```

- Or, without PL/SQL:

```
exec print_contact(50);
```

- Once a procedure is no longer needed, it can be removed with `drop procedure`:

```
drop procedure print_contact;
```

- It is also possible to infer a row type using `sys_refcursor` and return rows with `dbms_sql.return_result`:

```
create or replace procedure
    get_customer_by_credit(min_credit number)
as
    customer_cursor sys_refcursor;
begin
    open customer_cursor for select * from customers where credit_limit > min_credit;

    dbms_sql.return_result(customer_cursor);
end;
```

- You can now call it:

```
exec get_customer_by_credit(50);
```

Functions

- Functions are similar, but require returning a value:

```
create or replace function
    get_total_sales_for_year(year_arg integer)
return number
is
    total_sales number := 0;
begin
    select sum(unit_price * quantity) into total_sales
    from order_items
    inner join orders using (order_id)
    where status = 'Shipped'
    group by extract(year from order_date)
    having extract(year from order_date) = year_arg;

    return total_sales;
end;
```

- You can call them from PL/SQL:

```
declare
    total_sales number := 0;
begin
    total_sales := get_total_sales_for_year(2017);

    dbms_output.put_line('Sales for 2017: ' || total_sales);
end;
```

- And remove them with `drop function`:

```
drop function get_total_sales_for_year;
```

Packages

- Packages can be used to group function "interfaces" and variables:

```
create or replace package order_management
as
    shipped_status constant varchar(10) := 'Shipped';
    pending_status constant varchar(10) := 'Pending';
    cancelled_status constant varchar(10) := 'Canceled';

    function get_total_transactions return number;
end order_management;
```

- You can now access the variables in the package with `.`:

```
begin
    dbms_output.put_line(order_management.shipped_status);
end;
```

- In order to use functions in a package, you then have to create a package body, implementing it:

```
create or replace package body order_management
as
    function get_total_transactions return number
    is
        total_transactions number;
    begin
        select sum(unit_price) into total_transactions from orders inner join order_ite

        return total_transactions;
    end;
end;
```

- You can now access the functions in the package with `.`:

```
select
    order_management.get_total_transactions() as total_transactions
```

```
from
    dual;
```

- And the same is possible from PL/SQL:

```
begin
    dbms_output.put_line(order_management.get_total_transactions());
end;
```

- You can drop a package with `drop package` and a package body with `drop package body`:

```
drop package body order_management;
drop package order_management;
```

Triggers

- Triggers follow a similar structure as procedures:

```
declare
-- declarations
begin
-- your logic
exception
-- exception handling
end;
```

- Using triggers, you can for example create a manual log after operations with `after update or delete on ...`:

```
create or replace trigger customers_audit_trigger
    after update or delete
    on customers
    for each row
declare
    transaction_type varchar2(10);
begin
    transaction_type := case
        when updating then 'update'
        when deleting then 'delete'
    end;

    insert into audits(
        table_name,
        transaction_name,
        by_user,
        transaction_date
    ) values (
        'customers',
```

```
        transaction_type,
        user,
        sysdate
    );
end;
```

- Thanks to `before update of ... on ...`, it is also possible to do more complex checks before inserting:

```
create or replace trigger customers_credit_trigger
    before update of credit_limit
    on customers
declare
    current_day number;
begin
    current_day := extract(day from sysdate);

    if current_day between 28 and 31 then
        raise_application_error(-20100, 'Locked at the end of the month');
    end if;
end;
```

- In combination with `when`, `new` (not available in `delete` statements) and `old` (not available in `insert` statements), it is also possible to check based on the previous & current values:

```
create or replace trigger customers_credit_limit_trigger
    before update of credit_limit
    on customers
    for each row
    when (new.credit_limit > 0)
begin
    if :new.credit_limit >= 2*:old.credit_limit then
        raise_application_error(-20101, 'The new credit cannot be more than double the
    end if;
end;
```

- Using `instead of` triggers and `returning ... into ...`, you can also use views to safely insert into multiple tables:

```
create or replace trigger create_customer_trigger
    instead of insert on customers_and_contacts
    for each row
declare
    current_customer_id number;
begin
    insert into customers(
        name,
        address,
```

```
        website,
        credit_limit
    ) values (
        :new.name,
        :new.address,
        :new.website,
        :new.credit_limit
    ) returning customer_id into current_customer_id;

    insert into contacts(
        first_name,
        last_name,
        email,
        phone,
        customer_id
    ) values (
        :new.first_name,
        :new.last_name,
        :new.email,
        :new.phone,
        current_customer_id
    );
end;
```

- You can enable/disable a trigger with `alter trigger ... disable/enable`:

  ```
  alter trigger create_customer_trigger disable;
  ```

- And completely remove it with `drop trigger`:

  ```
  drop trigger create_customer_trigger;
  ```

- It is also possible to enable/disable all triggers of a table with `alter table ... enable/disable all triggers`:

  ```
  alter table customers enable all triggers;
  ```

Maps

- Maps are also possible in PL/SQL using `table of`:

  ```
  declare
      type country_capitals_type
          is table of varchar2(100)
          index by varchar2(50);

      country_capitals country_capitals_type;
  begin
      country_capitals('China') := 'Beijing';
  ```

```
    country_capitals('EU') := 'Brussels';
    country_capitals('USA') := 'Washington';
end;
```

- You can use `mymap.first` and `mymap.next` to iterate:

```
declare
    type country_capitals_type
        is table of varchar2(100)
        index by varchar2(50);

    country_capitals country_capitals_type;
    current_country varchar2(50);
begin
    country_capitals('China') := 'Beijing';
    country_capitals('EU') := 'Brussels';
    country_capitals('USA') := 'Washington';

    current_country := country_capitals.first;

    while current_country is not null loop
        dbms_output.put_line(current_country || ': ' || country_capitals(current_countr

        current_country := country_capitals.next(current_country);
    end loop;
end;
```

Arrays

- Using `varray`, it is also possible to create arrays:

```
declare
    type names_type is varray(255) of varchar2(20) not null;

    names names_type := names_type('Albert', 'Jonathan', 'Judy');
begin
    dbms_output.put_line('Length before append: ' || names.count);

    names.extend;

    names(names.last) := 'Alice';

    dbms_output.put_line('Length after append: ' || names.count);

    names.trim;

    dbms_output.put_line('Length after trim: ' || names.count);
```

```
    names.trim(2);

    dbms_output.put_line('Length after second trim: ' || names.count);

    names.delete;

    dbms_output.put_line('Length after delete: ' || names.count);
end;
```