

---

## **Uni DB1 Syntax Details**

Syntax details for the DB1 (databases) course at HdM  
Stuttgart

Felicitas Pojtinger

2022-02-01

## Inhaltsverzeichnis

<b>1</b>	<b>Acknowledgements</b>	<b>3</b>
<b>2</b>	<b>Reset Everything</b>	<b>3</b>
<b>3</b>	<b>SQL</b>	<b>3</b>
3.1	Operators . . . . .	3
3.2	Joins . . . . .	4
3.3	Aliases . . . . .	6
3.4	Limits and Pagination . . . . .	6
3.5	Dates and Intervals . . . . .	7
3.6	Expressions . . . . .	8
3.7	Grouping and Ordering . . . . .	9
3.8	Counting and Sums . . . . .	10
3.9	Inserting . . . . .	10
3.10	Switches . . . . .	11
3.11	Helper Functions . . . . .	11
3.12	Auto-Generated Primary Keys . . . . .	12
3.13	Modifying Columns . . . . .	12
3.14	Virtual Columns . . . . .	13
3.15	Modifying Tables . . . . .	13
3.16	Constraints . . . . .	14
3.17	Types . . . . .	15
3.18	Views . . . . .	15
3.19	Indexes . . . . .	16
<b>4</b>	<b>PL/SQL</b>	<b>16</b>
4.1	Block Structure . . . . .	16
4.2	Variables . . . . .	17
4.3	Fetching Data . . . . .	18
4.4	Branches and Expressions . . . . .	18
4.5	Switches . . . . .	19
4.6	Labels and Goto . . . . .	19
4.7	Loops . . . . .	20
4.8	Types and Objects . . . . .	21
4.9	Exceptions . . . . .	21
4.10	Cursors . . . . .	22
4.11	Locks . . . . .	23

---

4.12 Procedures . . . . .	23
4.13 Functions . . . . .	24
4.14 Packages . . . . .	25
4.15 Triggers . . . . .	26
4.16 Maps . . . . .	29
4.17 Arrays . . . . .	29

“so basically i am monkey” - monke, *monkeeee*

## 1 Acknowledgements

Most of the following is based on the [Oracle Tutorial](#).

## 2 Reset Everything

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

```
1 begin
2   for i in (select index_name from user_indexes where index_name not
3             like '%_PK') loop
4     execute immediate 'drop index ' || i.index_name;
5   end loop;
6   for i in (select trigger_name from user_triggers) loop
7     execute immediate 'drop trigger ' || i.trigger_name;
8   end loop;
9   for i in (select view_name from user_views) loop
10    execute immediate 'drop view ' || i.view_name;
11  end loop;
12  for i in (select table_name from user_tables) loop
13    execute immediate 'drop table ' || i.table_name || ' cascade
14                      constraints';
15  end loop;
16  execute immediate 'purge recyclebin';
17 end;
```

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

## 3 SQL

### 3.1 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 =, >, <.
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

### 3.2 Joins

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

```
1 select a.id as id_a, a.color as color_a, b.id as id_b, b.color as  
   color_b from palette_a a inner join palette_b b on a.color = b.  
   color;
```

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

```
1 select a.id as id_a, a.color as color_a, b.id as id_b, b.color as  
   color_b from palette_a a left join palette_b b on a.color = b.  
   color
```

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

```
1 select a.id as id_a, a.color as color_a, b.id as id_b, b.color as  
   color_b from palette_a a left join palette_b b on a.color = b.  
   color where b.id is null
```

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

```
1 select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_a a right join palette_b b on a.color = b.color;
```

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

```
1 select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_a a right join palette_b b on a.color = b.color where a.id is null;
```

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

```
1 select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_a a full join palette_b b on a.color = b.color;
```

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

```
1 select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_a a full join palette_b b on a.color = b.color where a.id is null or b.id is null;
```

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

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

- You can also use multiple on or using statements:

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

- If you use the on keyword, use **and** for multiples!

- You can also create the Cartesian product:

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

- It is also possible to do a self join:

```
1 select (w.first_name || ' ' || w.last_name) "Worker", (m.first_name || ' ' || m.last_name) "Manager", w.job_title from employees w left join employees m on w.employee_id = m.manager_id
```

- 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:

```
1 select first_name, last_name, email, 'contact' as role from
   contacts union select first_name, last_name, email, 'employee'
   as role from employees order by role
```

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

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

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

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

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

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

### 3.3 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.

### 3.4 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`:

```
1 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`:

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

- Need Pagination? Use offset:

```
1 select * from products order by standard_cost desc offset 10 rows
  fetch next 10 rows only;.
```

### 3.5 Dates and Intervals

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

```
1 select * from orders where status = 'Shipped' and extract(year
  from order_date) = 2017 order by order_date desc fetch next 1
  rows with ties;
```

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

```
1 select current_date from dual;
```

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

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

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

```
1 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:

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

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

```
1 select localtimestamp from dual;
```

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

```
1 select localtimestamp from dual;
```

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

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

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

```
1 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):

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

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

```
1 select interval '9' day from dual, select interval '9' month from
   dual, select interval '9-2' year to month from dual or select
   interval '09:08:6.75' hour to second(2) from dual;
```

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

### 3.6 Expressions

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

```
1 select product_name as "Product Name", list_price - standard_cost as "
   Gross Profit" from products order by "Gross Profit"
```

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

```
1 select * from orders where (
2 status = 'Canceled' or status = 'Pending' ) and customer_id = 44
3 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:

```
1 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`.

### 3.7 Grouping and Ordering

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

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

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

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

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

```
1 select name, count(*) as "Shipped Orders" from orders inner join
  customers using(customer_id) where status = 'Shipped' group by
  name order by "Shipped Orders" desc;
```

- `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:

```
1 select status from orders where extract(year from order_date) > '
  2016' group by status having status like '%d';
```

- 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).

### 3.8 Counting and Sums

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

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

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

```
1 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):

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

### 3.9 Inserting

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

```
1 insert into discounts(discount_name, amount, start_date,  
   expired_date) values ('Summer Promotion', 9.5, date '2017-05-01',  
   date '2017-08-31')
```

- You can also “insert from select” using `insert into`:

```
1 insert into sales(customer_id, product_id, order_date, total)  
   select customer_id, product_id, order_date, sum(quantity *  
   unit_price) amount from orders inner join order_items using(  
   order_id) where status = 'Shipped' group by customer_id,  
   product_id, order_date;
```

- 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):

```
1 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.):

```
1 insert all into fruits (fruit_name, color) values ('Apple', 'Red')
  into fruits (fruit_name, color) values ('Orange', 'Orange')
  into fruits (fruit_name, color) values ('Banana', 'Yellow')
select 1 from dual
```

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

```
1 insert all when amount < 10000 then into small_orders when amount
  >= 10000 then into big_orders select order_id, customer_id, (
  quantity * unit_price) amount from orders inner join
  order_items using (order_id)
```

### 3.10 Switches

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

```
1 select product_name, list_price, case category_id when 1 then
  round(list_price * 0.05, 2) when 2 then round(list_price * 0.1,
  2) else round(list_price * 0.2, 2) end discount from products
```

- `case` is also useful for conditional grouping:

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

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

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

### 3.11 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):

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

- You can use `replace` to replace strings:

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

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

### 3.12 Auto-Generated Primary Keys

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

```
1 create table persons ( person_id number generated by default as
  identity, first_name varchar2(50) not null, last_name varchar2
  (50), primary key(person_id) );
```

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

### 3.13 Modifying Columns

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

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

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

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

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

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

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

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

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

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

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

```
1 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:

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

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

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

- You can now drop it using:

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

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

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

### 3.14 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):

```
1 alter table parts add (capacity_description as ( case when
    capacity <= 8 then 'Small' when capacity > 8 then 'Large' end )
);
```

- 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.):

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

### 3.15 Modifying Tables

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

```
1 drop table people;
```

- Appending `purge` clears the recycle bin; appending `cascade constraints` drop all related constraints.

- You can clear a table using `truncate table`:

```
1 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:

```
1 purge recyclebin;
```

### 3.16 Constraints

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

```
1 alter table purchase_orders add constraint
  purchase_orders_order_id_pk primary key(order_id);
```

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

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

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

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

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

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

- You can also add foreign key constraints:

```
1 alter table suppliers add constraint suppliers_supplier_groups_fk
  foreign key(group_id) references supplier_groups(group_id);
```

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

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

- A unique constraint prevents unwanted duplicates:

```
1 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:

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

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

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

### 3.17 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 off 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.

### 3.18 Views

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

```
1 create view employees_years_of_service as select employee_id,
  first_name || ' ' || last_name as full_name, floor(
  months_between(current_date, hire_date) / 12) as
  years_of_service from employees;
```

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

```
1 create or replace view employees_years_of_service as select
  employee_id, first_name || ' ' || last_name as full_name, floor(
  months_between(current_date, hire_date) / 12) as
  years_of_service from employees with read only;
```

- `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:

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

- They can also be used in updates:



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

### 3.19 Indexes

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

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

- You can also create an index spanning multiple columns:

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

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

```
1 drop index members_full_name;
```

## 4 PL/SQL

### 4.1 Block Structure

- Block structure:

```
1 declare  
2 -- declarations  
3 begin  
4 -- your logic  
5 exception  
6 -- exception handling  
7 end;
```

- The most simple example is as follows:

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

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

```
1 declare  
2     message varchar(255) := 'Hello, World!';  
3 begin  
4     dbms_output.put_line(message);
```

```
5 end;
```

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

```
1 declare
2     result number;
3 begin
4     result := 1/0;
5
6     exception
7         when zero_divide then
8             dbms_output.put_line(sqlerrm);
9         when others then
10            dbms_output.put_line('An unexpected error occurred: '
11                                || sqlerrm);
12 end;
```

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

```
1 declare
2 begin
3     null;
4 end;
```

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

## 4.2 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:

```
1 declare
2     total_sales number(15,2);
3     credit_limit number(10,0);
4     contact_name varchar2(255);
5 begin
6     null;
7 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:

```
1 declare
2     shipping_status varchar2(25) not null := 'shipped';
3 begin
```

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

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

```
1 declare
2     price constant number := 10;
3 begin
4     price := 20; -- Will throw an exception
5 end;
```

### 4.3 Fetching Data

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

```
1 declare
2     customer_name customers.name%TYPE;
3     customer_credit_limit customers.credit_limit%TYPE;
4 begin
5     select
6         name, credit_limit
7     into
8         customer_name, customer_credit_limit
9     from customers where customer_id = 38;
10
11     dbms_output.put_line(customer_name || ': ' ||
12                          customer_credit_limit);
12 end;
```

### 4.4 Branches and Expressions

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

```
1 declare
2     sales number := 20000;
3 begin
4     if sales > 10000 then
5         dbms_output.put_line('Lots of sales!');
6     end if;
7 end;
```

- Inline expressions are also supported:

```
1 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:

```
1 declare
2     sales number := 20000;
3 begin
4     if sales > 10000 then
5         if sales > 15000 then
6             dbms_output.put_line('A new sales record!');
7         else
8             dbms_output.put_line('Lots of sales!');
9         end if;
10    end if;
11 end;
```

## 4.5 Switches

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

```
1 declare
2     grade char(1);
3     message varchar2(255);
4 begin
5     grade := 'A';
6
7     case grade
8         when 'A' then
9             message := 'Excellent';
10        when 'B' then
11            message := 'Great';
12        when 'C' then
13            message := 'Good';
14        when 'D' then
15            message := 'Fair';
16        when 'F' then
17            message := 'Poor';
18        else
19            raise case_not_found;
20        end case;
21
22    dbms_output.put_line(message);
23 end;
```

## 4.6 Labels and Goto

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

```
1  begin
2      goto do_work;
3      goto goodbye;
4
5      <<do_work>>
6      dbms_output.put_line('mawahaha');
7
8      <<goodbye>>
9      dbms_output.put_line('Goodbye!');
10 end;
```

## 4.7 Loops

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

```
1  declare
2      i number := 0;
3  begin
4      loop
5          i := i + 1;
6
7          dbms_output.put_line('Iterator: ' || i);
8
9          if i >= 10 then
10             exit;
11         end if;
12     end loop;
13
14     dbms_output.put_line('Done!');
15 end;
```

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

```
1  begin
2      for i in 0..100 loop
3          dbms_output.put_line(i);
4      end loop;
5  end;
```

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

```
1  declare
2      i number := 0;
3  begin
4      while i <= 100 loop
5          dbms_output.put_line(i);
6
7          i := i + 1;
```

```
8     end loop;
9 end;
```

## 4.8 Types and Objects

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

```
1 declare
2     customer customers%ROWTYPE;
3 begin
4     select * into customer from customers where customer_id = 100;
5     dbms_output.put_line(customer.name || '/' || customer.website)
6         ;
7 end;
```

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

```
1 declare
2     person persons%ROWTYPE;
3
4 begin
5     person.person_id := 1;
6     person.first_name := 'John';
7     person.last_name := 'Doe';
8
9     insert into persons values person;
10 end;
```

## 4.9 Exceptions

- You can create custom exceptions:

```
1 declare
2     e_credit_too_high exception;
3     pragma exception_init(e_credit_too_high, -20001);
4 begin
5     if 10000 > 1000 then
6         raise e_credit_too_high;
7     end if;
8 end;
```

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

```
1 declare
2     e_credit_too_high exception;
3     pragma exception_init(e_credit_too_high, -20001);
```

```
4 begin
5     raise_application_error(-20001, 'Credit is too high!');
6 end;
```

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

## 4.10 Cursors

- Using cursors, you can procedurally process data:

```
1 declare
2     cursor sales_cursor is select * from sales;
3     sales_record sales_cursor%ROWTYPE;
4 begin
5     update customers set credit_limit = 0;
6
7     open sales_cursor;
8
9     loop
10         fetch sales_cursor into sales_record;
11         exit when sales_cursor%NOTFOUND;
12
13         update
14             customers
15             set
16                 credit_limit = extract(year from sysdate)
17             where
18                 customer_id = sales_record.customer_id;
19     end loop;
20
21     close sales_cursor;
22 end;
```

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

```
1 declare
2     cursor product_cursor is select * from products;
3 begin
4     for product_record in product_cursor loop
5         dbms_output.put_line(product_record.product_name || ': $'
6                               || product_record.list_price);
7     end loop;
8 end;
```

- Cursors can also have parameters:

```
1 declare
2     product_record products%rowtype;
3     cursor
```

```
4      product_cursor (
5          low_price number := 0,
6          high_price number := 100
7      )
8  is
9      select * from products where list_price between low_price
10         and high_price;
11 begin
12     open product_cursor(50, 100);
13     loop
14         fetch product_cursor into product_record;
15         exit when product_cursor%notfound;
16
17         dbms_output.put_line(product_record.product_name || ' : $'
18                               || product_record.list_price);
19     end loop;
20     close product_cursor;
21 end;
```

#### 4.11 Locks

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

```
1 declare
2     cursor customers_cursor is select * from customers for update
3       of credit_limit;
4 begin
5     for customer_record in customers_cursor loop
6         update customers set credit_limit = 0 where customer_id =
7           customer_record.customer_id;
8     end loop;
9 end;
```

#### 4.12 Procedures

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

```
1 create or replace procedure
2     print_contact(customer_id_arg number)
3  is
4     contact_record contacts%rowtype;
5 begin
6     select * into contact_record from contacts where customer_id =
7       customer_id_arg;
```



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

- These procedures can then be executed:

```
1  begin
2      print_contact(50);
3  end;
```

- Or, without PL/SQL:

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

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

```
1  drop procedure print_contact;
```

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

```
1  create or replace procedure
2      get_customer_by_credit(min_credit number)
3  as
4      customer_cursor sys_refcursor;
5  begin
6      open customer_cursor for select * from customers where
                             credit_limit > min_credit;
7
8      dbms_sql.return_result(customer_cursor);
9  end;
```

- You can now call it:

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

## 4.13 Functions

- Functions are similar, but require returning a value:

```
1  create or replace function
2      get_total_sales_for_year(year_arg integer)
3  return number
4  is
5      total_sales number := 0;
6  begin
7      select sum(unit_price * quantity) into total_sales
8      from order_items
```

```
9     inner join orders using (order_id)
10    where status = 'Shipped'
11    group by extract(year from order_date)
12    having extract(year from order_date) = year_arg;
13
14    return total_sales;
15 end;
```

- You can call them from PL/SQL:

```
1 declare
2     total_sales number := 0;
3 begin
4     total_sales := get_total_sales_for_year(2017);
5
6     dbms_output.put_line('Sales for 2017: ' || total_sales);
7 end;
```

- And remove them with `drop` function:

```
1 drop function get_total_sales_for_year;
```

## 4.14 Packages

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

```
1 create or replace package order_management
2 as
3     shipped_status constant varchar(10) := 'Shipped';
4     pending_status constant varchar(10) := 'Pending';
5     cancelled_status constant varchar(10) := 'Canceled';
6
7     function get_total_transactions return number;
8 end order_management;
```

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

```
1 begin
2     dbms_output.put_line(order_management.shipped_status);
3 end;
```

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

```
1 create or replace package body order_management
2 as
3     function get_total_transactions return number
4     is
```

```
5      total_transactions number;  
6      begin  
7          select sum(unit_price) into total_transactions from orders  
              inner join order_items using(order_id);  
8  
9          return total_transactions;  
10     end;  
11 end;
```

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

```
1 select  
2     order_management.get_total_transactions() as  
        total_transactions  
3 from  
4     dual;
```

- And the same is possible from PL/SQL:

```
1 begin  
2     dbms_output.put_line(order_management.get_total_transactions()  
        );  
3 end;
```

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

```
1 drop package body order_management;  
2 drop package order_management;
```

## 4.15 Triggers

- Triggers follow a similar structure as procedures:

```
1 declare  
2 -- declarations  
3 begin  
4 -- your logic  
5 exception  
6 -- exception handling  
7 end;
```

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

```
1 create or replace trigger customers_audit_trigger  
2     after update or delete  
3     on customers
```

```
4   for each row
5 declare
6   transaction_type varchar2(10);
7 begin
8   transaction_type := case
9       when updating then 'update'
10      when deleting then 'delete'
11   end;
12
13   insert into audits(
14       table_name,
15       transaction_name,
16       by_user,
17       transaction_date
18   ) values (
19       'customers',
20       transaction_type,
21       user,
22       sysdate
23   );
24 end;
```

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

```
1 create or replace trigger customers_credit_trigger
2   before update of credit_limit
3   on customers
4 declare
5   current_day number;
6 begin
7   current_day := extract(day from sysdate);
8
9   if current_day between 28 and 31 then
10      raise_application_error(-20100, 'Locked at the end of the
11         month');
12   end if;
13 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:

```
1 create or replace trigger customers_credit_limit_trigger
2   before update of credit_limit
3   on customers
4   for each row
5   when (new.credit_limit > 0)
6 begin
7   if :new.credit_limit >= 2*old.credit_limit then
8      raise_application_error(-20101, 'The new credit cannot be
```

```
9         more than double the old credit!');
10     end if;
11 end;
```

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

```
1  create or replace trigger create_customer_trigger
2      instead of insert on customers_and_contacts
3      for each row
4  declare
5      current_customer_id number;
6  begin
7      insert into customers(
8          name,
9          address,
10         website,
11         credit_limit
12     ) values (
13         :new.name,
14         :new.address,
15         :new.website,
16         :new.credit_limit
17     ) returning customer_id into current_customer_id;
18
19     insert into contacts(
20         first_name,
21         last_name,
22         email,
23         phone,
24         customer_id
25     ) values (
26         :new.first_name,
27         :new.last_name,
28         :new.email,
29         :new.phone,
30         current_customer_id
31     );
32 end;
```

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

```
1  alter trigger create_customer_trigger disable;
```

- And completely remove it with `drop trigger`:

```
1  drop trigger create_customer_trigger;
```

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

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

## 4.16 Maps

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

```
1 declare
2     type country_capitals_type
3         is table of varchar2(100)
4         index by varchar2(50);
5
6     country_capitals country_capitals_type;
7 begin
8     country_capitals('China') := 'Beijing';
9     country_capitals('EU') := 'Brussels';
10    country_capitals('USA') := 'Washington';
11 end;
```

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

```
1 declare
2     type country_capitals_type
3         is table of varchar2(100)
4         index by varchar2(50);
5
6     country_capitals country_capitals_type;
7     current_country varchar2(50);
8 begin
9     country_capitals('China') := 'Beijing';
10    country_capitals('EU') := 'Brussels';
11    country_capitals('USA') := 'Washington';
12
13    current_country := country_capitals.first;
14
15    while current_country is not null loop
16        dbms_output.put_line(current_country || ': ' ||
17                               country_capitals(current_country));
18        current_country := country_capitals.next(current_country);
19    end loop;
20 end;
```

## 4.17 Arrays

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

```
1 declare
2     type names_type is varray(255) of varchar2(20) not null;
3
4     names names_type := names_type('Albert', 'Jonathan', 'Judy');
5 begin
6     dbms_output.put_line('Length before append: ' || names.count);
7
8     names.extend;
9
10    names(names.last) := 'Alice';
11
12    dbms_output.put_line('Length after append: ' || names.count);
13
14    names.trim;
15
16    dbms_output.put_line('Length after trim: ' || names.count);
17
18    names.trim(2);
19
20    dbms_output.put_line('Length after second trim: ' || names.
        count);
21
22    names.delete;
23
24    dbms_output.put_line('Length after delete: ' || names.count);
25 end;
```