# PL/SQL

Part 1: The basics

# JÜRGEN SIEBEN

ConDeS GmbH & Co. KG

j.sieben@condes.de

Github.com/j-sieben




Hear the full story

- Training on premise
- Workshops
- Code Review
- Team Coaching

# WHAT IS PL/SQL?

- Extension to SQL
  - Shares many of the syntactic details with SQL
  - Is executed within the database kernel, just where the data is
  - Is part of the database objects owned by a database user (schema)
- Turing complete programming language
  - Control structures (Loop, if-then-else etc.)
  - Scalar and complex variables
  - Possibility to store programs within the database (procedure, function, package, trigger etc.)

# WHAT IS PL/SQL USED FOR?

- As an extension to SQL, PL/SQL is mainly used to:
  - Create a table API
  - Grant access to database functionality not accessible via SQL
  - Extend SQL with user defined functionality
  - Perform repetitive administration tasks
- SQL is preferred to the usage of PL/SQL
- PL/SQL is preferred to the usage of external programming languages

# WHEN SHOULD PL/SQL NOT BE USED?

- If a problem is solvable with SQL, do it in SQL
  - Dig into the possibilities of SQL of your actual database version
  - Many efforts have been taken to avoid having to code functionality manually
  - If something is possible with SQL, it is faster and the Total Cost of Ownership is vastly reduced
- PL/SQL is not a fixture for bad database design
  - Good database design is a fixture for bad database design
  - Elaborate on decision tables and relation based programming techniques

# PL/SQL BLOCK

- PL/SQL is built using blocks
- Two mandatory and two optional keywords
  - `declare` (opt.)
  - `begin` (mand.)
  - `exception` (opt.)
  - `end` (mand.)
- All keywords are case insensitive
- All variable names share the same limitations as database object names in SQL

# PL/SQL BLOCK

```
declare
  l_ename varchar2(20 byte);
begin
  select ename
    into l_ename
    from emp
   where empno = 7389;
exception
  when others then
    raise;
end;
```

Starts declaration part of the block.
Here, variables, types and other objects can be declared for later use.

Starts execution part of the block.
Here, the coding takes place. PL/SQL blocks mix SQL and PL/SQL commands to create the desired functionality.

If anything goes wrong, control switches to the exception block.

If an exception occurs during execution of the code, controls switches to the exception block.
Any exception may be handled here, but this step is optional. If no handler exists, the exception is thrown to the calling environment.

All variables, types etc. need to be declared before they are allowed to be used.

# PL/SQL SYNTAX BASICS

- All keywords and commands are case insensitive
- There is no distinction between SQL and PL/SQL commands syntactically
- Every command is concluded by a semicolon
- In SQL*Plus, as with SQL (semicolon), the PL/SQL editor mode can only be left by entering a single / on a new line
- The beforementioned character is not part of PL/SQL but required when writing SQL scriptfiles including PL/SQL

# PL/SQL SYNTAX BASICS

- A PL/SQL block may occur in different shapes
  - Anonymous block
  - Stored procedure, stored function, package or trigger
  - As part of an object oriented type, implementing member functions
- Any PL/SQL block other than an anonymous block is part of the data dictionary and stored within system tables, like any other database object
- Oracle maintains a dependency chain, invalidating PL/SQL blocks when referenced database objects are changed

# PL/SQL SYNTAX BASICS

- A PL/SQL block may be nested into a surrounding `begin` – `end` block

```
begin
  <do_something>
  begin
    <do_something_else>
  exception
    <handle_errors>
  end;
end;
```

- This is useful to catch exceptions without having to leave the whole block

# SQL AND PL/SQL INTERACTION

- It's not easy to see where SQL ends and PL/SQL starts:

```
select ename
  into l_ename
  from emp
 where empno = l_empno;
```

- Any SQL function has it's PL/SQL counterpart, defined at `sys.standard`
- Exceptions:
  - `decode`
  - Any group functions, analytic functions etc.

# SQL AND PL/SQL INTERACTION

- Only DML (`insert`, `update`, `delete`, `merge`), `commit`, `rollback` and `select` statements are allowed within PL/SQL directly
- All other SQL statements may be executed dynamically by using `execute immedate '<sql statement>';`
- Obviously, all object privileges are obeyed, even when using dynamic SQL, so you need to be granted the necessary execution rights when using SQL
- SQL and PL/SQL are implemented within the database in two distinct areas
  - This allows for all SQL functionality to be used on the PL/SQL side without having to implement them twice
  - Disadvantage is that the database needs to switch between SQL and PL/SQL (aka context switch). More on that later.

# VARIABLES

- Any SQL type is a valid type for a PL/SQL variable
  - Scalar types like `varchar2`, `number`, `date` etc.
  - Referential types like `CLOB`, `BLOB` etc.
  - Object types like `XMLType`, `DICOM`, nested tables and objects
- Some types are PL/SQL specific
  - `boolean`, Records, PL/SQL tables (records and PL/SQL tables discussed in part 2)
  - `cursor` (SQL uses implicit cursors, but PL/SQL allows explicit control)
- Some types have different definitions
  - `varchar2` and `raw`: 32Kbyte in PL/SQL, 4000 (2000) byte in SQL

# VARIABLES

- Variables are defined using the same syntax as a column in a table definition

  ```
  <Name> <Type> [<Constraints>]
  ```

- Variables may be initialized with a value using keyword `default` or `:=`

  ```
  l_ename varchar2(20 byte) default 'KING';
  l_ename varchar2(20 byte) := 'KING';
  ```

- A constant is declared using the keyword constant and by assigning a value

  ```
  c_start_date constant date := date '2019-01-01';
  ```

- In rare occurrences, a variable may be defined outside the scope of PL/SQL (fi. In triggers). Then, the variable needs to be referenced with a colon-prefix:

  ```
  :new.ename := :old.ename;
  ```

# VARIABLES

- Variables can be bound to database columns or other variable declarations using %TYPE

```
l_empno emp.empno%TYPE;
l_interest number(5,2);
c_min_interest constant l_interest%TYPE := 1.15;
```

- Very useful to document and stabilize your code
  - You don't have to know the actual specification the variable will fit anyway
  - If you change the definition of the table column, the variable will change accordingly
  - If you bind the type to a table column, you document which information is to be expected within that variable
- Use this approach whenever possible, even if it's more code to write

# VARIABLES

- It's common to utilize a prefix system and underscores
  - Variables are named after database columns often but shall not have the same name
  - As PL/SQL is case insensitive, CamelCase is not a save choice
- It's allowed to have the same name for a variable and a column, but the following will not work and is therefore not recommended:

# VARIABLES

```
declare
  empno emp.empno%type;
  ename emp.ename%type;
begin
  select ename
    into ename  -- this is possible
    from emp
   where empno = empno; -- evaluates to true for any row
end;
```

# VARIABLES

- In PL/SQL, a boolean variable exists
- Allowed values are

  `true` | `false` | `null`
- The allowed values are assigned either directly or as a boolean expression

```
l_flag boolean := true;
l_flag boolean := sysdate > date '2019-01-01';
```

- In the `begin` block, a new value may be assigned to a variable using `:=`

# CONTROL STRUCTURES

- Control structures are divided into two groups
  - Decision trees
    - `if – elsif – else – end if;`
    - Simple and searched `case` expressions, alike SQL
  - Loops
    - Basic loop
    - `for` Loop
    - `while` loop
    - `cursor for` loop (discussed in part 2)

# CONTROL STRUCTURES – DECISION TREE

- if decision tree

```
if <expression> then
  <true tree>
elsif <expression> then
  <second true tree>
else
  <false tree>
end if;
```

# CONTROL STRUCTURES – DECISION TREE

- elsif is a known gotcha in PL/SQL
- `if` decision trees may be nested to form more complex decision trees
- `case` decision trees may be used as a replacement for complicated `if` decision trees
  - Simple case tree: Compare a variable with a given amount of values
  - Searched case tree: Provide each tree with a separate boolean expression
- As SQL does not support `if` decision trees, I prefer `case` decision trees except for trivial cases

# CONTROL STRUCTURES – DECISION TREE

- Simple case decision tree

```
case l_job
  when 'PRESIDENT' then l_sal := l_sal * 1.05;
  when 'MANAGER' then l_sal := l_sal * 1.03;
  else l_sal := l_sal * 1.04;
end case;
```

- Easy to understand
- Limited to simple equality comparisons

# CONTROL STRUCTURES – DECISION TREE

- Searched case decision tree

```
case
  when job = 'MANAGER‘ and dept = 10 then l_sal := l_sal * 1.02;
  when job = 'MANAGER' then l_sal := l_sal * 1.03;
  when job = 'PRESIDENT' then l_sal := l_sal * 1.05;
  else l_sal := l_sal * 1.04;
end case;
```

- More complex boolean expressions allowed
- Take care to place the most specific expressions first, `else` is the most generic

# CONTROL STRUCTURES – DECISION TREE

- `case` expressions can be nested, both simple and searched and in any combination
- Oracle reports that nesting `case` expressions is limited by memory only
- I report that nesting case expressions is limited to only very few levels to allow for proper understanding => KISS!
- Complex and long decision trees should be seen as a programming flaw that needs to be corrected
- Long decisions trees may be replaced by SQL and a proper data model, even dedicated decision tables may come into play here.

# CONTROL STRUCTURES – LOOPS

- Loops are used to execute a given code more than once
- Four types of loops
  - Basic loop: Require an exit condition to stop looping
  - `for` loop: Used if number of loop cycles is known upfront
  - `while` loop: Used if number of loop cycles depends on a dynamic condition
  - `cursor for` loop: Used to iterate over the result set of a SQL query (part 2)
- Decision between the loop types is based on maximum code clarity

# CONTROL STRUCTURES – LOOPS

- Basic loop

```
begin
  loop
    <do_something>;
    exit when <some condition>;
  end loop;
end;
```

- Use basic loop if both, a known number of executions and an unknown exit condition are possible

# CONTROL STRUCTURES – LOOPS

- `for` loop

```
for i in 1 .. l_amount loop
  <do_something>;
end loop;
```

- Use `for` loop if the number of loop cycles can be calculated before the loop starts
- Optional `reverse` keyword available to count from right to left hand side number
- If right hand side number is lower than left hand side number, no loop is executed
- If one of the numbers is `null`, an exception is thrown

# CONTROL STRUCTURES – LOOPS

- `while` loop

```
begin
  while <boolean expression> loop
    <do_something>;
  end loop;
end;
```

- Use `while` loop if the number of loop cycles is unknown before the loop starts
- Make sure that the boolean expression evaluates to `false` to stop the loop from cycling
- If the boolean condition is `false` upfront, no loop is executed

# CONTROL STRUCTURES - LOOPS

- It's very common to loop over a nested PL/SQL block

```
for i in 1 .. l_amount loop
  begin
    <do_something>
  exception
    <handle error>
  end;
end loop;
```

- This allows to proceed with the loop even in case of exceptions

# STORED PL/SQL BLOCKS

- PL/SQL block are stored within the database for easy reuse
- Use stored procedures or stored functions to encapsulate PL/SQL functionality
- Procedures and Functions are made even more useful by allowing to pass in parameters and to return one or more calculated values
- Bind PL/SQL to DML-events (insert, update, delete) to automatically execute them as a trigger
- Group related procedures and functions in larger building blocks called package

# STORED PL/SQL BLOCKS – PROCEDURES

- Encapsulate functionality within a database object
- Created by using the SQL command create
- `declare` is replaced with `is` or `as` to start the declaration part and is mandatory
- A procedure needs a name according to oracle database object name restrictions
- It may contain parameters
  - `in` parameters to pass information into the procedure
  - `out` parameters to get information out of the procedure
  - `in out` parameters to pass information into the procedure and return the calculated value using the same parameter

# STORED PL/SQL BLOCKS – PROCEDURES

```
SQL> create or replace procedure my_proc(
  2    p_param in varchar2)
  3  as
  4  begin
  5    dbms_output.put_line('Output: ' || p_param);
  6  end my_proc;
  7  /
Procedure created.
SQL> set serveroutput on
SQL> call my_proc('Hello world');
Output: Hello world
```

# STORED PL/SQL BLOCKS – PARAMETERS

- Parameters are directed (`in`, `out`, `in out`), defaulting to `in`
- Parameter type declaration references base datatypes only, no length or precision is given
- `out` and `in out` parameters create a shadow copy of the parameter values before executing the procedure to adhere to ACID standards
- If the shadow copy can be ommitted, parameters allow the hint `nocopy` to achieve this
- A parameter may become an optional parameter by assigning a default value

# STORED PL/SQL BLOCKS – PARAMETERS

- When calling a procedure or function, parameters can be passed in in various formats
  - Positional: All parameter values must be passed in in the same order as they are defined
  - Named: Any parameter can be referenced explicitly by providing the parameter name and value using the operator =>
- This is especially interesting in combination with optional parameters, meaning a parameter with a predefined default value

# STORED PL/SQL BLOCKS – PARAMETERS

```
create or replace procedure create_department(
  p_dname in dept.dname%type default 'unknown',
  p_loc in dept.loc%type default 'unknown')
as
begin
  insert into deptno(deptno, dname, loc)
  values (dept_seq.nextval, p_dname, p_loc);
end create_department;
```

# STORED PL/SQL BLOCKS – PARAMETERS

- Procedure create_department may be called using various syntax

```
call create_department('SALES‘, 'OHIO‘); -- positional
call create_department(
  p_loc => 'OHIO',
  p_dname => 'SALES'); -- named
call create_department(); -- using optional parameter values
call create_department('SALES'); -- p_loc optional
call create_department(p_loc => 'OHIO'); -- p_dname optional
```

# STORED PL/SQL BLOCKS – FUNCTIONS

- A function differs from a procedure in that it has to return one single value using the `return` clause
- The type of the return value is declared with the function declaration
- Functions are the only PL/SQL blocks that allow SQL functionality to be extended
- In order to make a function callable from SQL, make sure that the return value is a valid SQL datatype and that no `out` parameter is defined

# STORED PL/SQL BLOCKS – FUNCTIONS

```
create or replace function get_employee_name(
  p_empno in emp.empno%type)
  return emp.ename%type
as
  l_ename emp.ename%type;
begin
  select ename
    into l_ename
    from emp
   where empno := p_empno;
  return l_ename;
end get_employee_name;
```

# STORED PL/SQL BLOCKS – FUNCTIONS

- Two ways to call the function
  - PL/SQL:

    ```
    call dbms_output.put_line(get_employee_name(7839));
    ```

  - SQL:

    ```
    select get_employee_name(7389) ename
      from dual;
    ```

- All other rules apply as with procedures
  - Optional parameters
  - `in out` and `out` parameters (though strongly not recommended!)

# STORED PL/SQL BLOCKS – TRIGGERS

- A trigger is an anonymous PL/SQL block that gets executed if a DML event happens on a database table (simplified)
- The execution of a trigger may be once per DML statement or once per affected row of the DML statement
- If the trigger is executed once per affected row, it has access to the old and new status of the row (if this is logically possible)
- The trigger may be defined to execute before or after the event

# STORED PL/SQL BLOCKS – TRIGGERS

- To define a trigger, the following has to be decided:
  - Which database object is the trigger attached to?
  - On which events should the trigger react?
  - Does it fire before or after the event has occurred?
  - Does it fire once per statement or once per affected row?
- Access to the old an new row values is granted by using the pseudo variables `old` and `new`
- These pseudovariables are defined on the SQL side of the trigger and have therefore to be referenced as `:new` and `:old` within the PL/SQL block

# STORED PL/SQL BLOCKS – TRIGGERS

```
create or replace trigger trg_emp_briu
before insert or update on emp
for each row
begin
  :new.ename := upper(:new.ename);
  if updating then
    :new.empno := :old.empno;
  end if;
end;
```

# STORED PL/SQL BLOCKS – TRIGGERS

- As more than one trigger may be defined per table, use a naming convention that allows to see when the trigger fires
- Use a prefix to mark the database object as trigger to avoid name clashing with indexes, constraints or views on the same table
- Try to avoid triggers whenever possible in favour of a table API built as a package
- Keep in mind that switching between SQL and PL/SQL incurs a context switch, so a row trigger is not a valid approach in high speed applications

# PACKAGES

- A package is a collection of procedures, functions and other PL/SQL objects
- It offers a range of advantages over simple stored procedures/functions:
  - Distinction between publicly available objects and private helper methods
  - Possibility to overload the same method with more than one parameter footprint
  - Definition of datatypes
- Definition of a package is a two step process
  - Public interface
  - Implementation

# PACKAGES

- Public interface (aka specification)

```
create or replace package emp_pkg
as
  procedure fire_emp(
    p_empno in emp.empno%type);
end emp_pkg;
```

# PACKAGES

- Implementation (aka body)

```
create or replace package body emp_pkg
as
  procedure fire_emp(
    p_empno in emp.empno%type)
  as
  begin
    delete from emp
     where empno = p_empno;
  end fire_emp;
end emp_pkg;
```

# PACKAGES

- Only the specification is required, the implementation is optional
- Any method that is defined at the specification is public
- Methods with the same name must differ in their parameter footprint
- The specification and the body may contain methods, variables and type definitions
- A variable declaration outside a method makes it a global package variable, either public or private

# tbc

- By Chris!

- Thanks for your time