1 Declaring PL/SQL Variables

✅Recognize valid and invalid identifiers

Indentifiers: 2.2.2 Reserved words and keywords: D References to identifiers: 2.4

Identifiers

Identifiers name PL/SQL elements, which include:

-	-	-	-	-
constants	cursors	exceptions	keywords	types
labels	packages	reserved keywords	subprograms	variables

Every character in an identifier, alphabetic or not, is significant. e.g. 'last_name' and 'lastname' are different. Except in the case of quoted identifiers, PL/SQL is case-insensitive for identifiers, e.g. lastname, LastName, and LASTNAME are the same.

Reserved words and **keywords** are identifiers that have special meaning in PL/SQL. You cannot use reserved words as user-defined identifiers but you can use them as quoted user-defined identifiers, however this is not recommended. You can use keywords as ordinary user-defined identifiers, but it is not recommended.

Predefined identifiers are declared in the package STANDARD For a list of these identifiers, connect as a DBA and query: `select type_name from all_types where predefined='YES';`

A **user-defined** identifier is:

- composed of characters from the database character set
- either ordinary or quoted

An ordinary user-defined identifier:

- begins with a letter
- can include letters, digits, and these symbols:
  - dollar sign ($)
  - number sign (#)
  - underscore (_)
- is not a reserved word

A quoted user-defined identifier:

- is enclosed in double quotation marks
- allows any character form the database character set except double quotation marks, new line characters, and null characters.
- is case-sensitive, with one exception: if quoted would be valid ordinary without quotation marks, then the quotations are optional and if you do omit them then the identifier is case-insensitive

- can use reserved word as a quoted user-defined identifier, but is not recommended and you must always enclose the identifier in double quotation marks and it is always case-sensitive

Identifier length:

- if COMPATIBLE is set to 12.2 or higher, the representation of the identifier in the database character set cannot exceed 128 bytes (excl. double quote marks in quoted identifiers).
- if COMPATIBLE is set to 12.1 or lower, the limit is 30 bytes

References to identifiers

When referencing an identifier, you use an name that is either simple, qualified, remote, or both qualified and remote.

- the **simple name** of an identifier is the name in its declaration, e.g.

```
declare
  a integer; --declaration
begin
  a := 1; --reference with simple name
end;
/
```

- if identifier is declared in a named PL/SQL unit, you can (and sometimes must) refernce it with its **qualified name**, syntax (called **dot notation**) is: `unit_name.simple_identifier_name` e.g. if package p declares identifier a, you can reference the identifier with qualified name `p.a`
- if the identifier names an object on a remote database, you must reference it with its **remote name**: `simple_identifier_name@link_to_remote_database`
- **qualified remote name**: `unit_name.simple_identifier_name@link_to_remote_database`

You can create synonyms for remote schema object, but you cannot create synonyms for objects declared in PL/SQL subprograms or packages. You can reference identifiers declared in the packages STANDARD and DBMS_STANDARD without qualifying them with the package names, unless you have declard a local identifier with the same name.

✅List the uses of variables, declare and initialize variables, use bind variables

Declarations: 2.3 - 2.3.4 Scalar variable declaration syntax Constant declaration syntax Name Resolution: B Assigning values to variables: 2.6 VARIABLE command for bind variables Using bind variables: sqpug 5.12

Declarations

A declaration allocates storage space for a value of a specified data type, and names the storage location so that you can reference it. You must declare object before you can reference them. Declaration can appear in the declarative part of any block, subprogram, or package.

- NOT NULL Constraint
    - you can impose the NOT NULL constraint on a scalar variable or constant
    - the NOT NULL constraitn prevents assigning a null value to the item. Item can acquire this constraint either implicitly (from datatype, e.g. NATURALN, POSITIVEN, SIMPLE_INTEGER) or explicitly.
    - if scalar variable declaration specifies NOT NULL, then you must assign an initial value to the variable (because default initial value for scalar variable is NULL)
    - PL/SQL treats any zero-length string as a NULL value (includes values returned by character function and BOOLEAN expressions)
- Variable declaration
    - always specifies the name and data type of the variable and allocates storage for it
    - for most data types, can also specify an initial value, this value can change
    - variable must be a valis user-defined identifier
    - datatype can be any PL/SQL datatype which include SQL datatypes. Datatype is either scalar (without internal components) or composite (with internal components)
    - syntax: `variable datatype [[not null] {:= | default} expression];`
- Constant declaration
    - constant holds value that does not change
    - same rules for variables apply for constants, but constant declaration has two more requirements:
        - keyword CONSTANT
        - initial value of the constant e.g. `max_days_year constant integer := 365;`
    - syntax: `constant_identifier constant datatype [not null] {:= | default} expression;`
- Initial values of variables and constants
    - in variable declaration the initial value is optional unless you specify the not null constraint
    - in constant declaration the initial value is required
    - if declaration is in a block or subprogram, the initial value is assigned to the variable or constant every time control passes to the block or subprogram
    - if declaration is in a package specification, the initial value is assigned to the variable or constant for each session (whether public or private)
    - to specify initial value, use either the assignment operator (:=) or the keyword DEFAULT followed by an expression, which can include previously declared constants and initialized variables
    - if you do not specify an initial value for a variable, assign a value to it before using it in any other context

Name resolution

- Qualified names and dot notation
    - when one named item belongs to another named item, you can (and sometimes must) qualify the name of the "child" item with the name of the "parent" item, using dot notation. e.g.

sequence_name.currval
    - if an identifier is declared in a named unit, you can qualify its simple name (name in declaration) with the name of the unit (block, subprogram, or package), using syntax: `unit_name.simple_identifier_name`
    - if identifier not visible within scope then you must qualify its name
    - if identifier belongs to another schema, then you must qualify its name with the name of the schema, using syntax: `schema_name.package_name`

- Column name precedence
    - if a SQL statement references a name that belongs to both a column and either a local variable or formal parameter, then the column name takes precedence
    - > Caution: when a variable or parameter name is interpreted as a column name, data can be deleted, changed, or inserted unintentionally

- Differences between PL/SQL and SQL name resolution rules
    - PL/SQL rules are less permissive than SQL rules (most SQL rules are context-sensitive, are legal in more situations)
    - PL/SQL and SQL resolve qualified names differently, as example take table name HR.JOBS:
        - PL/SQL searches first for packages, types, tables, and views named HR in the current schema, then for public synonyms, and finally for objects named JOBS in the HR schema
        - SQL searches first for objects named JOBS in the HR schema, and then for packages, types, tables, and views named HR in the current schema

> When the pl/sql compiler processes static sql statement, it sends that statement to the sql subsystem, which uses sql rules to resolve names in the statement

- Resolution of names in static SQL statements when PL/SQL compiler finds static SQL statement:

1. if the statement is a SELECT statement, the PL/SQL compiler removes the INTO clause
2. the plsql compiler sends the statement to the sql subsystem
3. the the sql subsystem checks the syntax of the statement
4. if the sql subsystem cannot resolve a name in the scope of the sql statement, then it sends the name vack to the plsql compiler. the name is called an escaped identifier
5. the plsql compiler tries to resolve the escaped identifier
6. if the compilation of the plsql unit succeeds, the plsql compiler generates the text of the regular sql statement that is equivalent to the static sql statement and stores that text with the generated computer code
7. at run time, the plsql runtime system invokes routines that parse, bind, and run the regular sql statement (the bind variables are the escaped identifiers)
8. if the statement is a select statement, the plsql runtime system stores the results in the plsql targets specified in the into cluase that the plsql compile removed in step 1.

- What is capture?
    - when a declaration or deifinition prevents the compiler from correctly resolving a reference in another scope, the declaration or definition is said to capture the reference.

- **outer capture** occurs when a name in an inner scope, which had resolved to an item in an inner scope, now resolves to an item in an outer scope (both plsql and sql are designed to prevent outer capture, dont have to be careful for it)
- **same-scope capture** occurs when a columns is added to one of two tables used in a join, and the new column has the same name as a column in the other table. When only one table had a column with that name, the name could appear in the join unqualified. Now to avoud same-scope capture, you must qualify the column name with the approproate table name, everywhere that the column name appears in the join.
- **inner capture** occurs when a name in an inner scope which had resolved to an item in an outer scope, now either resolves to an item in an inner scope or cannot be resolved. In the first case, the result might change. In the second case, an error occurs. Example:

```
create table tab1 (col1 number, col2 number); -- tab1 has column named
col2
insert into tab1 values (100, 10);
create table tab2 (col1 number); -- tab2 does not have column named
col2
insert into tab2 values (100);

create or replace procedure proc is
cursor c1 is select * from tab1 where exists (
    select * from tab2 where col2 = 10
    -- tab2 has no col2, so reference to col2 resolves to tab1.col2
);
begin open c1; close c1; end;
/
-- if we add column named col2 to tab2 then the procedure becomes
invalid
-- it will be automatically recompiled at next invocation, and
reference will resolve to tab2
```

- Avoiding inner capture in select and dml statements
  - specify a unique alias for each table in the statement
  - do not specify a table alias that is the name of a schema that owns an item referenced in the statement
  - qualify each column reference in the statement with the appropriate table alias
    - to reference an attribute or method of a table element, you must give the table an alias and use the alias to qualify the reference too the attribute or method
    - row expressions must resolve as references to table aliases. A row expression can appear in the set clause of an update statement or be the parameter of the sql function ref or value

Assigning values to variables

After declaring a variable, you can assign a vaue to it in these ways:

- use the assignment statement to assign it the value of an expression
    - syntax: `assignment_statement_target := expression;`
    - assignment_statement_target:

```
{ collection_variable [index]   |
  cursor_variable               |
  :host_cursor_variable         | -- name of cursor var declared in
host env and passed as bind var
  object [.attribute]           | -- abstract datatype
  out_parameter                 |
  placeholder                   |
  record_variable [.field]      |
  scalar_variable
}
```

    - placeholder: `:host_variable [:indicator_variable]`
- use the select into or fetch statement to assign it a value from a table
    - simple syntax:

```
select select_item [, select_item...]...
into variable_name [, variable_name...]...
from table_name;
-- for each select_item there must be a corresponding, type-compatible
variable_name
-- because no boolean type in sql, variable_name cannot be boolean
```

- pass it to a subprogram as an out or in out parameter, and then assign the value inside the subprogram
    - if you pass a variable to a subprogram as an out or in out parameter, and the subprogram assigns a value to the parameter, the variable retains that value after the subprogram finishes running

> Variable and value must have compatible datatypes. One datatype is compatible with another data type if it can be implicitly converted to that type. The only values you can assign to a boolean variable are true, false, and null.

Bind variables

- bind variables are variables you create in sql*plus and then reference in plsql or sql
- if you create a bind variable in sql*plus you can use the variable as you would a declared variable in your plsql subprogram and then access the variable from sql*plus
- you can use a bind variable as an input bind variable to hold data which can then be used in plsql or sql statements to insert data into the database
- you can assign a value to a newly defined variable, which can then be used in a statement

- Creating bind variables
    - you create bind variables in sql*plus with the `VARIABLE` command
    - syntax: `var[iable] [variable [type [=value]]]` e.g. `variable ret_val number = 12`
    - `variable` without arguments displays a list of all the variables declared in the session
    - `variable` followed only by a variable name lists that variable
    - bind variables may be used as parameters to stored procedures or directly referenced in anonymous blocks
    - to display value of bind variable created with variable, use the `print [variable ...]` command,
    - to auto display the value of a bind variable, use `SET AUTOP[RINT] {ON | OFF}`
    - bind variables cannot be used in the `copy` command or sql statements, except in plsql blocks, instead use substitution variables
    - when you execute `variable ... clob or nclob`, sqlplus associates a lob locator with the bind variable. The lob locator is auto populated when you execute a `select clob_column into :cv` statement in a plsql block. sqlplus closes the lob locator when you exit sqlplus.
    - to free resources used by clob and nclob bind variables you may need to manually free temporary LOBs with: `execute dbms_lob.freetemporary(:cv)`, all temp lobs are freed when you exit sqlplus
    - sqlplus `set` commands such as `set long` and `set longchunksize` and `set loboffset` may be used to control the size of the buffer while printing clob or nclob bind variables
    - when you execute `variable ... refcursor` command, sqlplus creates a cursor bind variable. The cursor is auto opened by an `open ... for select` statement referencing the bind variable in a plsql block. sqlplus closes the cursor after completing a `print` statement for that bind variable or on exit
    - a refcursor bind variable may not be `print`ed more than once without re-executing the plsql `open ... for` statement

- Referencing bind variables
    - you reference bind variables in plsql by typing a colon :, followed immediately by the name of the variable

```
variable ret_val number;
begin
  :ret_val := 4;
end;
/
PL/SQL procedure successfully completed.
```

- Diplaying bind variables
    - use the sqlplus `PRINT` command

```
print ret_val
RET_VAL
```

```
----------
 4
```

- Executing an input bind
    - you can assign a value for input binding

```
variable x number = 123
select :x from dual;
        :X
----------
        123

insert into emp(id) values (:x);
1 row created.
```

✅List and describe various data types using the %TYPE and %ROWTYPE attributes

PL/SQL datatypes: 3 Abstract data types (ADT): 1.2.8.5 Declaring items using %TYPE: 2.3.5 %TYPE Declaring items using %ROWTYPE: 5.12.4 %ROWTYPE

Data types

Every plsql constant, variable, parameter, and function return value has a **data type** that determines its storage format and its valid values and operations. **Scalar datatypes** store values with no internal components. A scalar data type can have subtypes. A **subtype** is a data type that is a subset of another data type, which is its **base type**. A subtype has the same valid operations as its base type. A data type and its subtypes comprise a **data type family**. PL/SQL predefines many types and subtypes in the package `STANDARD` and lets you define your own subtypes. The plsql scalar datatypes are: sql datatypes, boolean, pls_integer, binary_integer, ref cursor, user-defined subtypes

- SQL data types
    - unlike sql, plsql lets you declare variables
    - some sql datatypes have different maximum sizes in plsql

data type	size in SQL	size in PL/SQL
char, nchar, raw	2000 bytes	32767 bytes
varchar2, nvarchar2	4000 bytes	32767 bytes
long	2 GB - 1	32760 bytes
long raw	2 GB	32760 bytes
blob, clob, nclob	(4 GB - 1)*db_block_size	128 TB

    - binary_float and binary_double computations do not raise exceptions so you must check the values that they produce for conditions such as overflow and underflow by comparing them to predefined constants Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants
    - plsql has additional subtypes of binary_float and binary_double:
        - simple_float (subtype of binary_float) and simple_double (subtype of binary_double)
        - each of these has the same range as its base type and has a not null constraint
        - if you know that variable will never have value null, then declare it as these subtypes rather than the base types. Without the overhead of checking nullness, the subtypes provide significantly better performance than their base types.
    - char and varchar2 variables
        - if the value you assign to a character variable is longer than the maximum size of the variable, an error occurs (numeric or value error: character string buffer too small)
        - same if you insert character variable into column and value of variable is longer than the defined width of the column, error (value too large for column ... (actual: 5, maximum: 3))
        - char has one predefined subtype in both sql and plsql => character
        - varchar2 has one predefined subtypes in both => varchar, and one extra predefined subtype in plsql only => string
    - long and long raw variables
        - you can insert any long, char or varchar2 value into a long column and any raw or long raw value into a long raw column
        - you cannot retrieve a value longer than 32760 bytes from a long or long raw column into a long or long raw variable, same for char or varchar2 variables but with values longer than 32767 bytes
    - rowid and urowid variables
        - when you retrieve a rowid into a rowid variable, use the `rowidtochar` function to convert the binary value to character value
        - to convert the value of rowid variable to rowid, use the `chartorowid` function
        - if value does not represent a valid rowid, plsql raises predefined exception SYS_INVALID_ROWID
        - to retrieve a rowid into a urowid variable, or to convert value of urowid variable to rowid, use an assignment statement; the conversion in implicit

- Boolean data type
    - the plsql datatype boolean stores **logical values**, which are the boolean values true and false and the value null, null represents unknown value
    - syntax: `variable_name BOOLEAN`
    - because sql has no data type equivalent to boolean, you cannot:
        - assign a boolean value to a database table column
        - select or fetch the value of a database table column into a boolean variable
        - use a boolean value in a sql function (but can invoke plsql function with boolean parameter, however the argument cannot be a boolean literal, workaround: assign the

literal to a variable and then pass the variable to the function)
        - use a boolean expression in a sql statement, except as an argument to a plsql function invoked in a sql query or in an anonymous block
    - cannot pass a boolean value to the dbms_output.put[_line] subprograms, to print boolean use if or case statement and translate to character

- PLS_INTEGER and BINARY_INTEGER data types
    - pls_integer and binary_integer are identical (moving forward, both are implicated when one is mentioned)
    - pls_integer stores signed integers in range -2,147,483,648 through 2,147,483,647 in 32 bits
    - pls_integer has advantages over the number data type and number subtypes:
        - pls_integer values require less storage
        - pls_integers operations use hardware arithmetic, so they are faster than number operations, which use library arithmetic. For efficiency use pls_integer values for all calculations in its range
    - a calculation with two pls_integer values that overflows the pls_integer range raises an overflow exception (even if the result is assigned to number datatype). For calculations outside this range, use integer (as one of the calculation values), a predefined subtype of the number datatype
    - pls_integer has predefined subtypes:
        - natural (nonnegative), naturaln (nonnegative with not null constraint)
        - positive (positve), positiven (positive not null)
        - signtype (-1, 0, 1)
        - simple_integer (not null)
    - pls_integer and its subtypes can be implicitly converted to: char, varchar2, number, long all of those except long can be implicitly converted to pls_integer
    - pls_integer value can be implicitly converted to on of its subtypes only if the value does not violate a constraint of the subtype
    - simple integer differs from pls_integer in its overflow semantics, if you know that a variable will never have null or need overflow checking, declare it as simple_integer, performs significantly better than pls_integer
    - in simple_integer overflows are ignored so value can wrap from positive to negative or vice-versa
    - integer literals in the simple_integer range have the datatype simple_integer

- User-defined plsql subtypes
    - plsql lets you define your own subtypes, the base type can be any scalar or user-defined plsql data type specifier such as char, date or record (including previously user-defined subtype)
    - subtypes can
        - provide compatibility with anso/iso data types
        - show the intended use of data items of that type
        - detect out of range values
    - an **unconstrained subtype** has the same set of values as its vase type, so it is only another name for the base type, therefore if they are of the same base type then they are interchangeable with each other and with the base type, no data type conversion occurs syntax: `subtype subtype_name is base_type`, e.g. `subtype "DOUBLE PRECISION" is float` (ansi compatible)

    - a **constrained subtype** has only a subset of the values of its base type. If the base type lets you specify size, precision and scale, or a range of values, then you can specify them for its subtypes. syntax: `subtype subtype_name is base_type {precision [, scale] | range low_value .. high_value} [not null]` a constrained subtype can be implicitly converted to its base type, but the base type can be implicitly converted to the constrained subtype only if the value does not violate a constraint of the subtype same goes for converting constrained subtype to another constrained subtype with same base type, can only happen if source value does not violate constraint of the target
    - if two subtypes have different base types in the same data type family, then one subtype can be implicitly converted to the other only if the source value does not violate a constraint of the target subtype.

For a list of datatypes and datatype families in the STANDARD package, see PL/SQL predefined data types

Abstract data types

- An Abstract Data Type (ADT) consists of a data structure and subprograms that manipulate the data
- The variables that form the data structure are called **attributes**. The subprograms that manipulate the attributes are called **methods**
- ADTs are stored in the database. Instances of ADTs can be stored in tables and used as PL/SQL variables
- ADTs let you reduce complexity by separating a large system into logical components, which you can reuse
- In the static data dictionary view `*_OBJECTS`, the `OBJECT_TYPE` of an ADT is `TYPE`. In the static data dictionary view `*_TYPES`, the `TYPECODE` of an ADT is `OBJECT`
- > Note: ADTs are also called user-defined types and object types

%TYPE

- the `%TYPE` attribute lets you declare a data item of the same datatype as a previously declared variable or column (without knowing what that type is)
- the item declared with `%type` is the **referencing item**, and the previously declared item is the **referenced item**
- if the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly
- the `%type` attribute cannot be used if th referenced character column has a collation other than `using_nls_comp`
- syntax: `referencing_item referenced_item%TYPE;` referenced_item can be: `collection_variable`, `cursor_variable`, `db_table_or_view.column`, `object`, `record_variable[.field]`, `scalar_variable`
- the referencing item inherits the following from the referenced item:
    - data type and size
    - constraints (unless the referenced item is a column)
- the referencing item does not inherit the initial value of the referenced item, therefore if the referencing item specifies or inherits the `NOT NULL` constraint, you must specify an initial value for it
- the `%TYPE` attribute is particularly useful when declaring variables to hold database values, declaring a variable of the same type as a column: `variable_name table_name.column_name%type`

%ROWTYPE

- the `%ROWTYPE` attribute lets you declare a **record variable** that represents either a full or partial row of a database table or view
- For every visible column of the full or partial row, the record has a field with the same name and data type. If the structure of the row changes, then the structure of the record changes accordingly. Making an invisible column visible changes the structure of some records declared with the %ROWTYPE attribute.
- to declare a record variable that always represents a full row of a database table or view, use this syntax: `variable_name table_or_view_name%rowtype;` for every column of the table or view, the record then has a field with the same name and data type
- `%rowtype` variable des not inherit initial values or constraints
- to declare a record variable that can represent a partial row of a database table or view, use syntax: `variable_name {explicit_cursor|cursor_variable}%rowtype` a cursor is associated with a query, for every column that the query selects, the record variable must have a corresponding type-compatible field. If the query selects every row then variable represents full row, otherwise variable represents partial row (can also be join row, join query). The cursor must be either an explicit cursor or a strong cursor variable
- if you use the `%rowtype` attribute to define a record variable that represents a full row of a table that has a virtual column, then you cannot insert that record into the table (*ORA-54013: INSERT operation disallowed on virtual columns*). Instead you must insert the individual record fields into the table, excluding the virtual column
- suppose that you use the `%rowtype` attribute to define a record variable that represents a row of a table that has an invisible column, and then you make the invisible column visible.
    - if you define the record variable with a cursor, then making the invisible column visible does not change the structure of the record variable
    - if you define the record variable as a full row and use a `select * into` statement to assign values to the record, then making the invisible column visible does change the structure of the record example: table t with columns a, b and invisible c declaring record as t%rowtype will have only fields a and b, not c (select * into will only fill a, b) altering the table and making c visible will make the record have field c at following usage