|  |
| --- |
| , RD Dep. |
| **Oracle PL/SQL** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial Version | Siarhei Kandrashevich | 15-Jan-2014 |  |  |
|  |  |  |  |  |  |

Contents

[1. Basics of PL/SQL 3](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645225)

[1.1. PL/SQL Language Fundamentals 3](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645226)

[1.1.1. PL/SQL Character Set 3](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645227)

[1.1.2. Identifiers 3](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645228)

[1.1.3. Boolean, Numeric, and String Literals 4](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645229)

[1.1.4. Datetime Interval Literals (Oracle9i) 4](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645230)

[1.1.5. Delimiters 4](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645231)

[1.1.6. Comments 5](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645232)

[1.1.7. Pragmas 6](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645233)

[1.1.8. Statements 6](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645234)

[1.1.9. Block Structure 7](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645235)

[1.2. Variables and Program Data 7](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645236)

[1.2.1. Scalar Datatypes 8](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645237)

[1.2.2. Numeric datatypes 8](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645238)

[1.2.3. Character datatypes 9](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645239)

[1.2.4. Datetime datatypes 10](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645240)

[1.2.5. Boolean datatype 11](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645241)

[1.2.6. NLS Character Datatypes 12](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645242)

[1.2.7. LOB Datatypes 12](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645243)

[1.2.8. NULLs in PL/SQL 13](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645244)

[1.2.9. Declaring Variables 13](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645245)

[1.2.10. Constrained declarations 13](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645246)

[1.2.11. Constants 13](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645247)

[1.2.12. Default values 14](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645248)

[1.2.13. Anchored Declarations 14](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645249)

[1.2.14. Programmer-Defined Subtypes 14](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645250)

[2. Nested blocks in PL/SQL 16](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645251)

[2.1. Conditional and Sequential Control 16](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645252)

[2.1.1. Conditional Control Statements 16](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645253)

[2.1.2. IF-THEN combination 16](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645254)

[2.1.3. IF-THEN-ELSE combination 16](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645255)

[2.1.4. IF-THEN-ELSIF combination 16](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645256)

[2.1.5. CASE statement 17](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645257)

[2.1.6. CASE expression 17](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645258)

[2.1.7. Sequential Control Statements 19](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645259)

[2.1.8. GOTO 19](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645260)

[2.1.9. NULL 19](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645261)

[2.2. Loops 20](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645262)

[2.2.1. Simple Loop 20](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645263)

[2.2.2. Numeric FOR Loop 20](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645264)

[2.2.3. Cursor FOR Loop 21](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645265)

[2.2.4. WHILE Loop 21](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645266)

[2.2.5. REPEAT UNTIL Loop Emulation 21](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645267)

[2.2.6. EXIT Statement 22](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645268)

[2.2.7. Loop Labels 22](file:///D:\Work\Students\Oracle%20SQL%20Introduction\Topic%20-%204\MTN.NIX.07.Oracle_SQL_courseware4_%20PLSQL1.docx#_Toc380645269)

# Basics of PL/SQL

## PL/SQL Language Fundamentals

This section summarizes the fundamental components of the PL/SQL language: characters, identifiers, literals, delimiters, use of comments and pragmas, and construction of statements and blocks.

### PL/SQL Character Set

The PL/SQL language is constructed from letters, digits, symbols, and whitespace, as defined in the following table:

|  |  |
| --- | --- |
| **Type** | **Characters** |
| Letters | A-Z, a-z |
| Digits | 0-9 |
| Symbols | ~!@#$%^&\*( )\_-+=|[ ]{ }:;"'< >,.?/ ^ |
| Whitespace | space, tab, newline, carriage return |

Characters are grouped together into four lexical units: identifiers, literals, delimiters, and comments.

### Identifiers

Identifiers are names for PL/SQL objects such as constants, variables, exceptions, procedures, cursors, and reserved words. Identifiers have the following characteristics:

* Can be up to 30 characters in length
* Cannot include whitespace (space, tab, carriage return)
* Must start with a letter
* Can include a dollar sign ($), an underscore ( \_ ), and a pound sign (#)
* Are not case-sensitive

In addition, you must not use PL/SQL's reserved words as identifiers. For a list of those words, see the table in the final section in this book.

If you enclose an identifier within double quotes, then all but the first of these rules are ignored. For example, the following declaration is valid:

DECLARE

"1 ^abc" VARCHAR2(100);

BEGIN

IF "1 ^abc" IS NULL THEN ...

END;

### Boolean, Numeric, and String Literals

Literals are specific values not represented by identifiers. For example, TRUE, 3.14159, 6.63E-34, `Moby Dick', and NULL are all literals of type Boolean, number, or string. There are no complex datatype literals as they are internal representations. Unlike the rest of PL/SQL, literals are case-sensitive. To embed single quotes within a string literal, place two single quotes next to each other. See the following table for examples:

|  |  |
| --- | --- |
| **Literal** | **Actual value** |
| 'That''s Entertainment!' | That's Entertainment! |
| '"The Raven"' | "The Raven" |
| 'TZ=''CDT6CST''' | TZ='CDT6CST' |
| '''' | ' |
| '''hello world''' | 'hello world' |
| '''''' | '' |

### Datetime Interval Literals (Oracle9i)

The datetime interval datatypes are new with Oracle9i. These datatypes represent a chronological interval expressed in terms of either years and months or days, hours, minutes, seconds, and fractional seconds. Literals of these datatypes require the keyword INTERVAL followed by the literal and format string(s). The interval must go from a larger field to a smaller one, so YEAR TO MONTH is valid, but MONTH TO YEAR is not. See the following table for examples:

|  |  |
| --- | --- |
| **Literal** | **Actual value** |
| INTERVAL `1-3' YEAR TO MONTH | 1 year and 3 months later |
| INTERVAL `125-11' YEAR(3) TO MONTH | 125 years and 11 months later |
| INTERVAL `-18' MONTH | 18 months earlier |
| INTERVAL `-48' HOUR | 48 hours earlier |
| INTERVAL `7 23:15' DAY TO MINUTE | 7 days, 23 hours, 15 minutes later |
| INTERVAL `1 12:30:10.2' DAY TO SECOND | 1 day, 12 hours, 30 minutes, 10.2 seconds later |
| INTERVAL `12:30:10.2' HOUR TO SECOND | 12 hours, 30 minutes,10.2 seconds later |

### Delimiters

Delimiters are symbols with special meaning, such as := (assignment operator), || (concatenation operator), and ; (statement delimiter). The following table lists the PL/SQL delimiters:

|  |  |
| --- | --- |
| **Delimiter** | **Description** |
| ; | Terminator (for statements and declarations) |
| + | Addition operator |
| - | Subtraction operator |
| \* | Multiplication operator |
| / | Division operator |
| \*\* | Exponentiation operator |
| || | Concatenation operator |
| := | Assignment operator |
| = | Equality operator |
| <> and != | Inequality operators |
| ^= and ~= | Inequality operators |
| < | "Less than" operator |
| <= | "Less than or equal to" operator |
| > | "Greater than" operator |
| >= | "Greater than or equal to" operator |
| ( and ) | Expression or list delimiters |
| << and >> | Label delimiters |
| , | (Comma) Item separator |
| ' | (Single quote) Literal delimiter |
| " | (Double quote) Quoted literal delimiter |
| : | Host variable indicator |
| % | Attribute indicator |
| . | (Period) Component indicator (as in record.field or package.element) |
| @ | Remote database indicator (database link) |
| => | Association operator (named notation) |
| .. | (Two periods) Range operator (used in the FOR loop) |
| -- | Single-line comment indicator |
| /\* and \*/ | Multiline comment delimiters |

### Comments

Comments are sections of the code that exist to aid readability. The compiler ignores them.

A single-line comment begins with a double hyphen (—) and ends with a new line. The compiler ignores all characters between the — and the new line.

A multiline comment begins with slash asterisk (/\*) and ends with asterisk slash (\*/). The /\* \*/ comment delimiters can also be used for a single-line comment. The following block demonstrates both kinds of comments:

DECLARE

-- Two dashes comment out only the physical line.

/\* Everything is a comment until the compiler

encounters the following symbol \*/

You cannot embed multiline comments within a multiline comment, so be careful during development if you comment out portions of code that include comments. The following code demonstrates this issue:

DECLARE

/\* Everything is a comment until the compiler

/\* This comment inside another WON'T work!\*/

encounters the following symbol. \*/

/\* Everything is a comment until the compiler

-- This comment inside another WILL work!

encounters the following symbol. \*/

### Pragmas

The PRAGMA keyword is used to give instructions to the compiler. There are four types of pragmas in PL/SQL:

*EXCEPTION\_INIT*

Tells the compiler to associate the specified error number with an identifier that has been declared an EXCEPTION in your current program or an accessible package

*RESTRICT\_REFERENCES*

Tells the compiler the purity level of a packaged program. The purity level is the degree to which a program does not read/write database tables and/or package variables.

*SERIALLY\_REUSABLE*

Tells the runtime engine that package data should not persist between references. This is used to reduce per-user memory requirements when the package data is only needed for the duration of the call and not for the duration of the session.

*AUTONOMOUS\_TRANSACTION*

Starting in Oracle8i, tells the compiler that the function, procedure, top-level anonymous PL/SQL block, object method, or database trigger executes in its own transaction space.

### Statements

A PL/SQL program is composed of one or more logical statements. A statement is terminated by a semicolon delimiter. The physical end-of-line marker in a PL/SQL program is ignored by the compiler, except to terminate a single-line comment (initiated by the — symbol).

### Block Structure

Each PL/SQL program is a block consisting of a standard set of elements, identified by keywords. The block determines the scope of declared elements, and how exceptions are handled and propagated. A block can be anonymous or named. Named blocks include functions, procedures, packages, and triggers.

Here is an example of an anonymous block:

DECLARE

today DATE DEFAULT SYSDATE;

BEGIN

-- Display the date.

DBMS\_OUTPUT.PUT\_LINE ('Today is ' || today);

END;

Here is a named block that performs the same action:

CREATE OR REPLACE PROCEDURE show\_the\_date

IS

today DATE DEFAULT SYSDATE;

BEGIN

-- Display the date.

DBMS\_OUTPUT.PUT\_LINE ('Today is ' || today);

END show\_the\_date;

The following table summarizes the sections of a PL/SQL block:

|  |  |
| --- | --- |
| **Section** | **Description** |
| Header | Required for named blocks. Specifies the way the program is called by other PL/SQL blocks. Anonymous blocks do not have a header. They start with the DECLARE keyword if there is a declaration section, or with the BEGIN keyword if there are no declarations. |
| Declaration | Optional; declares variables, cursors, TYPEs, and local programs that are used in the block's execution and exception sections. |
| Execution | Optional in package and TYPE specifications; contains statements that are executed when the block is run. |
| Exception | Optional; describes error-handling behavior for exceptions raised in the executable section. |

## Variables and Program Data

PL/SQL programs are normally used to manipulate database information. You commonly do this by declaring variables and data structures in your programs, and then working with that PL/SQL-specific data.

A variable is a named instantiation of a data structure declared in a PL/SQL block (either locally or in a package). Unless you declare a variable as a CONSTANT, its value can be changed at any time in your program.

The following table summarizes the different types of program data:

|  |  |
| --- | --- |
| **Type** | **Description** |
| Scalar | Variables made up of a single value, such as a number, date, or Boolean |
| Composite | Variables made up of multiple values, such as a record or a collection |
| Reference | Pointers to values |
| LOB | Variables containing large object (LOB) locators |

### Scalar Datatypes

Scalar datatypes divide into four families: number, character, datetime, and Boolean.

### Numeric datatypes

Numeric datatypes are further divided into decimal, binary integer, and PLS\_INTEGER storage types.

Decimal numeric datatypes store fixed and floating-point numbers of just about any size. They include NUMBER, DEC, DECIMAL, NUMERIC, FLOAT, REAL, and DOUBLE PRECISION. The maximum precision of a variable with type NUMBER is 38 digits, which yields a range of values from 1.0E-129 through 9.999E125. (This range of numbers would include the mass of an electron over the mass of the universe or the size of the universe in angstroms.)

Variables of type NUMBER can be declared with precision and scale, as follows:

NUMBER(precision, scale)

where precision is the number of digits, and scale is the number of digits to the right (positive scale) or left (negative scale) of the decimal point at which rounding occurs. Legal values for scale range from -84 to 127. The following table shows examples of precision and scale:

|  |  |  |
| --- | --- | --- |
| **Declaration** | **Assigned value** | **Stored value** |
| NUMBER | 6.02 | 6.02 |
| NUMBER(4) | 8675 | 8675 |
| NUMBER(4) | 8675309 | Error |
| NUMBER(12,5) | 3.14159265 | 3.14159 |
| NUMBER(12,-5) | 8675309 | 8700000 |

Binary integer numeric datatypes store whole numbers. They include BINARY\_INTEGER, INTEGER, INT, SMALLINT, NATURAL, NATURALN, POSITIVE, POSITIVEN, and SIGNTYPE. Binary integer datatypes store signed integers in the range of -231 + 1 to 231 - 1. The subtypes include NATURAL (0 through 231) and POSITIVE (1 through 231) together with the NOT NULL variations NATURALN and POSITIVEN. SIGNTYPE is restricted to three values (-1, 0, 1).

PLS\_INTEGER datatypes have the same range as the BINARY\_INTEGER datatype, but use machine arithmetic instead of library arithmetic, so are slightly faster for computation-heavy processing.

The following table lists the PL/SQL numeric datatypes with ANSI and IBM compatibility. In this table:

* prec is the precision for the subtype.
* scale is the scale of the subtype.
* binary is the binary precision of the subtype.

|  |  |  |
| --- | --- | --- |
| **PL/SQL datatype** | **Compatibility** | **Oracle RDBMS datatype** |
| DEC(prec,scale) | ANSI | NUMBER(prec,scale) |
| DECIMAL(prec,scale) | IBM | NUMBER(prec,scale) |
| DOUBLE PRECISION | ANSI | NUMBER |
| FLOAT(binary) | ANSI, IBM | NUMBER |
| INT | ANSI | NUMBER(38) |
| INTEGER | ANSI, IBM | NUMBER(38) |
| NUMERIC(prec,scale) | ANSI | NUMBER(prec,scale) |
| REAL | ANSI | NUMBER |
| SMALLINT | ANSI, IBM | NUMBER(38) |

### Character datatypes

Character datatypes store alphanumeric text and are manipulated by character functions. As with the numeric family, there are several subtypes in the character family, shown in the following table:

|  |  |
| --- | --- |
| **Family** | **Description** |
| CHAR | Fixed-length alphanumeric strings. Valid sizes are 1 to 32767 bytes (which is larger than the Oracle database limit of 4000). |
| VARCHAR2 | Variable-length alphanumeric strings. Valid sizes are 1 to 32767 bytes (which is larger than the Oracle database limit of 4000). |
| LONG | Variable-length alphanumeric strings. Valid sizes are 1 to 32760 bytes. LONG is included primarily for backward compatibility. CLOB is the preferred datatype for large character strings. |
| RAW | Variable-length binary strings. Valid sizes are 1 to 32767 bytes (which is larger than the Oracle database limit of 2000). RAW data do not undergo character set conversion when selected from a remote database. |
| LONG RAW | Variable-length binary strings. Valid sizes are 1 to 32760 bytes. LONG RAW is included primarily for backward compatibility. BLOB and BFILE are the preferred datatypes for large binary data. |
| ROWID | Fixed-length binary data. Every row in a database has a physical address or ROWID. A ROWID has four parts in base 64:  OOOOOOFFFBBBBBBRRR  where:  OOOOOO is the object number.  FFFF is the absolute or relative file number.  BBBBBBBB is the block number within the file.  RRRR is the row number within the block. |
| UROWID | Universal ROWID. Variable-length hexadecimal string depicting a logical, physical, or non-Oracle row identifier. Valid sizes are up to 4000 bytes. |

### Datetime datatypes

Oracle expanded support for datetime data in Oracle9i by introducing an assortment of new datatypes. The datetime datatypes are DATE (the only datetime datatype pre-Oracle9i), TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE. The two interval datatypes, also new to Oracle9i, are INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND.

DATE values are fixed-length, date-plus-time values. The DATE datatype can store dates from January 1, 4712 B.C. to December 31, 9999 A.D. Each DATE includes the century, year, month, day, hour, minute, and second. Sub-second granularity is not supported via the DATE datatype; use one of the TIMESTAMP datatypes instead. The time portion of a DATE defaults to midnight (12:00:00 AM) if it is not included explicitly.

TIMESTAMP values store date and time to sub-second granularity. The sub-second precision (the number of digits to the right of the decimal) either defaults or is set to 0 through 9 digits by declaration, as in:

DECLARE

mytime\_declared TIMESTAMP(9);

mytime\_default TIMESTAMP;

The default precision is 6 digits of precision to the right of the decimal.

TIMESTAMP WITH TIME ZONE values store date and time values like a TIMESTAMP but also store the hourly offset from UTC (Coordinated Universal Time, which is essentially equivalent to Greenwich Mean Time). As with TIMESTAMP, the sub-second precision is 0 to 9 digits, either declared or inherited from the default 6 digits of precision.

DECLARE

mytime\_declared TIMESTAMP(9) WITH TIME ZONE;

mytime\_default TIMESTAMP WITH TIME ZONE;

TIMESTAMP WITH LOCAL TIME ZONE values store date and time values together with the UTC offset, like a TIMESTAMP WITH TIME ZONE. The principal difference between these timestamp datatypes occurs when values are saved to or retrieved from a database table. TIMESTAMP WITH LOCAL TIME ZONE values are converted to the database time zone and saved without an offset. The values retrieved from the database table are converted from the database time zone to the session's time zone.

The offset from UTC for both TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE can be hours and minutes or a time zone region (found in the V$TIMEZONE\_NAMES data dictionary view) with the optional daylight savings time name (also found in V$TIMEZONE\_NAMES). For example:

ALTER SESSION SET NLS\_TIMESTAMP\_TZ\_FORMAT=

'DD-Mon-YYYY HH24:MI:SS.FF TZR';

DECLARE

my\_tswtz TIMESTAMP(4) WITH TIME ZONE;

BEGIN

my\_tswtz := '31-JUL-02 07:32:45.1234 US/Pacific';

INTERVAL YEAR TO MONTH values store a period of time in years and months:

DECLARE

myy2m INTERVAL YEAR TO MONTH;

BEGIN

myy2m := INTERVAL '1-6' YEAR TO MONTH;

INTERVAL DAY TO SECOND values store a period of time in days, hours, minutes, seconds, and fractional seconds:

DECLARE

myd2s INTERVAL DAY TO SECOND;

BEGIN

myd2s := INTERVAL '2 10:32:15.678' DAY TO SECOND;

### Boolean datatype

The BOOLEAN datatype can store one of only three values: TRUE, FALSE, or NULL. BOOLEAN variables are usually used in logical control structures such as IF...THEN or LOOP statements.

The following truth tables show the results of logical AND, OR, and NOT operations with PL/SQL's three-value Boolean model:

|  |  |  |  |
| --- | --- | --- | --- |
| **AND** | **TRUE** | **FALSE** | **NULL** |
| TRUE | TRUE | FALSE | NULL |
| FALSE | FALSE | FALSE | FALSE |
| NULL | NULL | FALSE | NULL |

|  |  |  |  |
| --- | --- | --- | --- |
| **OR** | **TRUE** | **FALSE** | **NULL** |
| TRUE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | NULL |
| NULL | TRUE | NULL | NULL |

|  |  |  |
| --- | --- | --- |
| **NOT (TRUE)** | **NOT (FALSE)** | **NOT (NULL)** |
| FALSE | TRUE | NULL |

### NLS Character Datatypes

The standard WE8MSWIN1252 or WE8ISO8859P2 character set does not support some languages, such as Chinese and Greek. To support a secondary character set, Oracle allows two character sets in a database—the database character set and the national character set (NLS).

The two NLS datatypes, NCHAR and NVARCHAR2, are used to represent data in the national character set. NCHAR values are fixed-length character data; the maximum length is 32767 bytes. NVARCHAR2 values are variable-length character data; the maximum length is also 32767 bytes.

### LOB Datatypes

PL/SQL supports a number of large object (LOB) datatypes, which can store objects of up to four gigabytes of data. Unlike the scalar datatypes, variables declared for LOBs use locators, or pointers to the actual data. LOBs are manipulated in PL/SQL using the built-in package DBMS\_LOB. The LOB datatypes are:

*BFILE*

File locators pointing to read-only large binary objects in operating system files. With BFILEs, the large objects are outside the database.

*BLOB*

LOB locators that point to large binary objects inside the database.

*CLOB*

LOB locators that point to large character (alphanumeric) objects inside the database.

*NCLOB*

LOB locators that point to large national character set objects inside the database.

### NULLs in PL/SQL

PL/SQL represents unknown or inapplicable values as NULL values. Because a NULL is unknown, a NULL is never equal or not equal to anything (including another NULL value). In addition, most functions return a NULL when passed a NULL argument—the notable exceptions are NVL, NVL2, CONCAT, and REPLACE. You cannot check for equality or inequality to NULL; therefore, you must use the IS NULL or IS NOT NULL syntax to check for NULL values.

Here is an example of the IS NULL syntax used to check the value of a variable:

BEGIN

IF myvar IS NULL

THEN

...

### Declaring Variables

Before you can use a variable, you must first declare it in the declaration section of your PL/SQL block or in a package as a global. When you declare a variable, PL/SQL allocates memory for the variable's value and names the storage location so that the value can be retrieved and changed. The syntax for a variable declaration is:

variable\_name datatype [CONSTANT] [NOT NULL]

[{ := | DEFAULT } initial\_value]

### Constrained declarations

The datatype in a declaration can be constrained or unconstrained. Constrained datatypes have a size, scale, or precision limit that is less than the unconstrained datatype. For example:

total\_sales NUMBER(15,2); -- Constrained.

emp\_id VARCHAR2(9); -- Constrained.

company\_number NUMBER; -- Unconstrained.

book\_title VARCHAR2; -- Not valid.

Constrained declarations require less memory than unconstrained declarations. Not all datatypes can be specified as unconstrained. You cannot, for example, declare a variable to be of type VARCHAR2. You must always specify the maximum size of a variable-length string.

### Constants

The CONSTANT keyword in a declaration requires an initial value and does not allow that value to be changed. For example:

min\_order\_qty NUMBER(1) CONSTANT := 5;

### Default values

Whenever you declare a variable, it is assigned a default value of NULL. Initializing all variables is distinctive to PL/SQL; in this way, PL/SQL differs from languages such as C and Ada. If you want to initialize a variable to a value other than NULL, you do so in the declaration with either the assignment operator (:=) or the DEFAULT keyword:

counter BINARY\_INTEGER := 0;

priority VARCHAR2(8) DEFAULT 'LOW';

A NOT NULL constraint can be appended to the variable's datatype declaration to indicate that NULL is not a valid value. If you add the NOT NULL constraint, you must explicitly assign an initial value for that variable.

### Anchored Declarations

Use the %TYPE attribute to anchor the datatype of a scalar variable to either another variable or to a column in a database table or view. Use %ROWTYPE to anchor a record's declaration to a cursor or table.

The following block shows several variations of anchored declarations:

DECLARE

tot\_sales NUMBER(20,2);

-- Anchor to a PL/SQL variable.

monthly\_sales tot\_sales%TYPE;

-- Anchor to a database column.

v\_ename employee.last\_name%TYPE;

CURSOR mycur IS

SELECT \* FROM employee;

-- Anchor to a cursor.

myrec mycur%ROWTYPE;

The NOT NULL clause on a variable declaration (but not on a database column definition) follows the %TYPE anchoring and requires anchored declarations to have a default in their declaration. The default value for an anchored declaration can be different from that for the base declaration:

tot\_sales NUMBER(20,2) NOT NULL DEFAULT 0;

monthly\_sales tot\_sales%TYPE DEFAULT 10;

### Programmer-Defined Subtypes

PL/SQL allows you to define unconstrained scalar subtypes. An unconstrained subtype provides an alias to the original underlying datatype; for example:

CREATE OR REPLACE PACKAGE std\_types

IS

-- Declare standard types as globals.

SUBTYPE dollar\_amt\_t IS NUMBER;

END std\_types;

CREATE OR REPLACE PROCEDURE process\_money

IS

-- Use the global type declared above.

credit std\_types.dollar\_amt\_t;

...

A constrained subtype limits or constrains the new datatype to a subset of the original datatype. For example, POSITIVE is a constrained subtype of BINARY\_INTEGER. The declaration for POSITIVE in the STANDARD package is:

SUBTYPE POSITIVE IS BINARY\_INTEGER RANGE 1..2147483647;

You can define your own constrained subtypes in your programs:

PACKAGE std\_types

IS

SUBTYPE currency\_t IS NUMBER (15, 2);

END;

# Nested blocks in PL/SQL

## Conditional and Sequential Control

PL/SQL includes conditional (IF, CASE) structures as well as sequential control (GOTO, NULL) constructs.

### Conditional Control Statements

There are several varieties of IF-THEN-ELSE and CASE structures.

### IF-THEN combination

IF condition THEN

executable statement(s)

END IF;

For example:

IF caller\_type = 'VIP' THEN

generate\_response('GOLD');

END IF;

### IF-THEN-ELSE combination

IF condition THEN

TRUE sequence\_of\_executable\_statement(s)

ELSE

FALSE/NULL sequence\_of\_executable\_statement(s)

END IF;

For example:

IF caller\_type = 'VIP' THEN

generate\_response('GOLD');

ELSE

generate\_response('BRONZE');

END IF;

### IF-THEN-ELSIF combination

IF condition-1 THEN

statements-1

ELSIF condition-N THEN

statements-N

[ELSE

ELSE statements]

END IF;

For example:

IF caller\_type = 'VIP' THEN

generate\_response('GOLD');

ELSIF priority\_client THEN

generate\_response('SILVER');

ELSE

generate\_response('BRONZE');

END IF;

### CASE statement

There are two types of CASE statements: simple and searched.

A simple CASE statement is similar to an IF-THEN-ELSIF structure. The statement has a switch expression immediately after the keyword CASE. The expression is evaluated and compared to the value in each WHEN clause. The first WHEN clause with a matching value is executed and then control passes to the next statement following the END CASE. For example:

CASE region\_id

WHEN 'NE' THEN

mgr\_name := 'MINER';

WHEN 'SE' THEN

mgr\_name := 'KOOI';

ELSE mgr\_name := 'LANE';

END CASE;

If a switch expression evaluates to NULL, the ELSE case is the only one that can possibly match; WHEN NULL will never match because Oracle performs an equality comparison on the expressions.

Both the CASE statement and the CASE expression (see the next section) should include an ELSE clause that will execute statements if no WHEN clause evaluates TRUE, because PL/SQL's runtime engine will raise an exception if it finds no matching expression.

The searched CASE statement does not have a switch; instead, each WHEN clause has a complete Boolean expression. The first matching WHEN clause is executed and control passes to the next statement following the END CASE. For example:

CASE

WHEN region\_id = 'EAME' THEN

mgr\_name := 'SCHMIDT';

WHEN division = 'SALES' THEN

mgr\_name := 'KENNEDY';

ELSE mgr\_name := 'GUPTA';

END CASE;

### CASE expression

There are also two types of CASE expressions: simple and searched. You can use CASE expressions anywhere that you can use any other type of expressions in PL/SQL programs.

A simple CASE expression lets you choose an expression to evaluate based on a scalar value that you provide as input. The following example shows a simple CASE expression being used with the built-in DBMS\_OUTPUT package to output the value of a Boolean variable. DBMS.OUTPUT.PUT\_LINE is not overloaded to handle Boolean types, so in this example the CASE expression converts the Boolean value in a character string, which PUT\_LINE can then handle:

DECLARE

boolean\_true BOOLEAN := TRUE;

boolean\_false BOOLEAN := FALSE;

boolean\_null BOOLEAN;

FUNCTION boolean\_to\_varchar2 (flag IN BOOLEAN)

RETURN VARCHAR2 IS

BEGIN

RETURN

CASE flag

WHEN TRUE THEN 'True'

WHEN FALSE THEN 'False'

ELSE 'NULL' END;

END;

BEGIN

DBMS\_OUTPUT.PUT\_LINE(boolean\_to\_varchar2(boolean\_true));

DBMS\_OUTPUT.PUT\_LINE(boolean\_to\_varchar2(boolean\_false));

DBMS\_OUTPUT.PUT\_LINE(boolean\_to\_varchar2(boolean\_null));

END;

A searched CASE expression evaluates a list of expressions to find the first one that evaluates to TRUE, and then returns the results of an associated expression. In the following example, a searched CASE expression returns the proper bonus value for any given salary:

DECLARE

salary NUMBER := 20000;

employee\_id NUMBER := 36325;

PROCEDURE give\_bonus

(emp\_id IN NUMBER, bonus\_amt IN NUMBER) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE(emp\_id);

DBMS\_OUTPUT.PUT\_LINE(bonus\_amt);

END;

BEGIN

give\_bonus(employee\_id,

CASE

WHEN salary >= 10000 AND salary <=20000 THEN 1500

WHEN salary > 20000 AND salary <= 40000 THEN 1000

WHEN salary > 40000 THEN 500

ELSE 0

END);

END;

### Sequential Control Statements

PL/SQL provides a GOTO statement and a NULL statement to aid in sequential control operations.

### GOTO

The GOTO statement performs unconditional branching to a named label. You should only rarely use a GOTO. At least one executable statement must follow the label (the NULL statement can be this necessary executable statement). The format of a GOTO statement is:

GOTO <<label\_name>>;

For example:

BEGIN

GOTO second\_output;

DBMS\_OUTPUT.PUT\_LINE('This line will never execute.');

<<second\_output>>

DBMS\_OUPUT.PUT\_LINE('We are here!);

END

There are a number of scope restrictions on where a GOTO can branch control. A GOTO:

* Can branch out of an IF statement, LOOP, or sub-block
* Cannot branch into an IF statement, LOOP, or sub-block
* Cannot branch from one section of an IF statement to another (from the IF-THEN section to the ELSE section is illegal)
* Cannot branch into or out of a sub-program
* Cannot branch from the exception section to the executable section of a PL/SQL block
* Cannot branch from the executable section to the exception section of a PL/SQL block, although a RAISE does this

### NULL

The NULL statement is an executable statement that does nothing. It is useful when an executable statement must follow a GOTO label or to aid readability in an IF-THEN-ELSE structure. For example:

IF :report.selection = 'DETAIL' THEN

exec\_detail\_report;

ELSE

NULL;

END IF;

## Loops

The LOOP construct allows you to execute a sequence of statements repeatedly. There are three kind of loops: simple (infinite), FOR, and WHILE.

You can use the EXIT statement to break out of LOOP and pass control to the statement following the END LOOP.

### Simple Loop

LOOP

executable\_statement(s)

END LOOP;

The simple loop should contain an EXIT or EXIT WHEN unless you want it to execute infinitely. Use the simple loop when you want the body of the loop to execute at least once. For example:

LOOP

FETCH company\_cur INTO company\_rec;

EXIT WHEN company\_cur%ROWCOUNT > 5 OR

company\_cur%NOTFOUND;

process\_company(company\_cur);

END LOOP;

### Numeric FOR Loop

FOR loop\_index IN [REVERSE] lowest\_number..highest\_number

LOOP

executable\_statement(s)

END LOOP;

The PL/SQL runtime engine automatically declares the loop index a PLS\_INTEGER variable; never declare a variable with that name yourself. The lowest\_number and highest\_number ranges can be variables, but are evaluated only once—on initial entry into the loop. The REVERSE keyword causes PL/SQL to start with the highest\_number and decrement down to the lowest\_number. For example, this code:

BEGIN

FOR counter IN 1 .. 4

LOOP

DBMS\_OUTPUT.PUT(counter);

END LOOP;

DBMS\_OUTPUT.NEW\_LINE;

FOR counter IN REVERSE 1 .. 4

LOOP

DBMS\_OUTPUT.PUT(counter);

END LOOP;

DBMS\_OUTPUT.NEW\_LINE;END;

yields the following output:

1234

4321

### Cursor FOR Loop

FOR record\_index IN [cursor\_name | (SELECT statement)]

LOOP

executable\_statement(s)

END LOOP;

The PL/SQL runtime engine automatically declares the loop index a record of cursor\_name%ROWTYPE; never declare a variable with that name yourself.

The cursor FOR loop automatically opens the cursor, fetches all rows identified by the cursor, and then closes the cursor. You can embed the SELECT statement directly in the cursor FOR loop. For example:

FOR emp\_rec IN emp\_cur

LOOP

IF emp\_rec.title = 'Oracle Programmer'

THEN

give\_raise(emp\_rec.emp\_id,30)

END IF;

END LOOP;

### WHILE Loop

WHILE condition

LOOP

executable\_statement(s)

END LOOP;

Use the WHILE loop in cases where you may not want the loop body to execute even once:

WHILE NOT end\_of\_analysis

LOOP

perform\_analysis;

get\_next\_record;

IF analysis\_cursor%NOTFOUND AND next\_step IS NULL

THEN

end\_of\_analysis := TRUE;

END IF;

END LOOP;

### REPEAT UNTIL Loop Emulation

PL/SQL does not directly support a REPEAT UNTIL construct, but a modified simple loop can emulate one. The syntax for this emulated REPEAT UNTIL loop is:

LOOP

executable\_statement(s)

EXIT WHEN Boolean\_condition;

END LOOP;

Use the emulated REPEAT UNTIL loop when executing iterations indefinitely before conditionally terminating the loop.

### EXIT Statement

EXIT [WHEN condition];

If you do not include a WHEN clause in the EXIT statement, it will terminate the loop unconditionally. Otherwise, the loop terminates only if the Boolean condition evaluates to TRUE. The EXIT statement is optional and can appear anywhere in the loop.

### Loop Labels

Loops can be optionally labeled to improve readability and execution control, as we showed earlier in the discussion of the GOTO statement. The label must appear immediately in front of the statement that initiates the loop.

The following example demonstrates the use of loop labels to qualify variables within a loop and also to terminate nested and outer loops:

<<year\_loop>>

FOR yearind IN 1 .. 20

LOOP

<<month\_loop>>

LOOP

...

IF year\_loop.yearind > 10

THEN

EXIT year\_loop;

END IF;

END LOOP month\_loop;

END LOOP year\_loop;

# 