آموزش برنامه نویسی

PL/SQL

برای دیتابیس اوراکل

میلاد خالقی -- 1399

فهرست مطالب

1 مقدمات PL/SQL

PL/SQL یک زبان برنامه نویسی برای دیتابیس اوراکل می باشد که توسط شرکت اوراکل طراحی و ارائه شده است. ساختار این زبان بر پایه بلاک(BLOCK) است بنابراین برنامه های PL/SQL به قسمت هایی از کد به نام بلاک تقسیم بندی می شوند. در این فصل اجزای یک بلاک و سایر مقدمات مورد نیاز برای فراگیری این زبان را توضیح می دهیم.

1.1 اجزای بلاک در PL/SQL

در PL/SQL هر بلاک از 3 بخش زیر تشکیل می شود:

1- DECLARATION: این بخش با کلمه کلیدی DECLARE شروع می شود. در این قسمت، متغیرها، زیربرنامه ها و هر عنصر دیگری که قرار است در داخل برنامه استفاده شوند تعریف می شوند. استفاده از DECLARE در بلاک ها، اختیاری است.

2- EXECUTABLE COMMANDS: این قسمت بین کلمه کلیدی BEGIN و END قرار می گیرد و شامل دستورات اجرایی آن برنامه می شود. استفاده از این بخش اجباری است ولی می توان بین BEGIN و END از دستور NULL استفاده کرد به این معنی که این بلاک هیچ کاری انجام نمی دهد.

3- EXCEPTION HANDLING: این قسمت با کلمه کلیدی EXCEPTION شروع می شود. اگر در حین اجرای برنامه خطایی رخ می دهد، آن برنامه متوقف شده و EXCEPTION HANDLER اجرا می شود. استفاده از این بخش اختیاری است.

نکته: دستورات PL/SQL با یک علامت **;** خاتمه می یابد.

نکته: بلاک های PL/SQL می توانند با استفاده از BEGIN و END درون بلاکهای دیگر استفاده شوند.

ساختار کلی یک بلاک PL/SQL به شکل زیر است:

```
DECLARE
      <declarations section>
BEGIN
      <executable command(s)>
EXCEPTION
```

```
      <exception handling>
END;
```

مثال: متن 'HELLO, World!' را در خروجی نمایش دهید.

```
DECLARE
      message varchar2(20):= 'Hello, World!';
BEGIN
      dbms_output.put_line(message);
END;
/
```

نکته: برای اجرای یک بلاک PL/SQL در محیط ابزار SQLPLUS یا در SQL COMMAND LINE باید از علامت **/ بعد** از پایان بلاک استفاده کنیم.

نکته: نام ها یا نشانگرهایی که در PL/SQL استفاده می شوند باید شرطهای زیر را داشته باشند:

- با حروف انگلیسی شروع شوند.
- کمتر از 30 کاراکتر باشند.
- می توان از اعداد ، حروف یا علامت $ ، _ یا # استفاده کرد.

1.2 DELIMETERها در PL/SQL

DELIMETER، یک علامت است که در زبان PL/SQL دارای کاربرد و مفهوم خاص است. در جدول زیر لیستی از DELIMETERهای زبان PL/SQL به همراه توضیح آنها را می بینید.

مفهوم	DELIMETER
عملیات ریاضی	+, -, *, /
نشانگر ATTRIBUTE	%
علامت نشانگر کاراکتر	'
انتخابگر اجزا	.
علامت جدا کننده	(,)
نشانگر متغیرهایHOST	:
جداکننده آیتم ها	,
نشانگر QUOTEDها	"
عملگر ارتباطی	=
نشانگر مربوط به دسترسی خارجی	@
علامت خاتمه دهنده دستورات	;
عملگر تخصیص دهنده	:=
عملگر مرتبط کننده	=>
عملگر الحاق	||
عملگر نمایی	**
نشانگرهای LABEL	<< , >>
COMMENT چندخطی	/* , */
COMMENT در یک خط	--
عملگر مربوط به رنج	..
عملگر ارتباطی	<, >, <=, >=
علامت نامساوی	<>, '=, ~=, ^=

1.3 توضیحات برنامه در زبان PL/SQL

در PL/SQL همانند سایر زبان های برنامه نویسی می توانیم توضیحاتی در مورد کد یا اجزای هر بلاک را به کد برنامه اضافه کنیم. این توضیحات می توانند با استفاده از علامت -- در یک خط باشند یا با استفاده از علامت های /* و */ به صورت چند خطی نوشته شوند. این قسمت ها در زمان اجرای دستورات، نادید گرفته می شوند.

مثال:

```
DECLARE
      -- variable declaration
      message varchar2(20):= 'Hello, World!';
BEGIN
      /*
       * PL/SQL executable statement(s)
       */
      dbms_output.put_line(message);
END;
/
```

نکته: هر کدام از موارد زیر را یک واحد PL/SQL یا PL/SQL UNIT می نامند:

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

1.4 نوع داده ها در PL/SQL

برای متغیرها، CONSTANTها یا پارامترهای استفاده شده در برنامه های PL/SQL می بایست یک نوع داده مناسب تعریف کرد تا فرمت صحیح برای ذخیره سازی داده و محدوده آنها مشخص گردد. این نوع داده ها معمولا به صورت SCALAR یا LOB هستند که آنها را در ادامه معرفی می کنیم.

نوع داده های SCALAR، مقدارهای واحد هستند که انواع آنها را در جدول زیر مشاهده می کنید.

نوع داده	توضیحات
عددی	**مقدارهای از نوع عددی که عملیات ریاضی روی آنها انجام می شود.**
کاراکتری	**مقدارهای از نوع حروف الفبا که به صورت کاراکتر یا رشته ای از کاراکترها هستند.**
BOOLEAN	**مقدارهای منطقی که عملیات منطقی روی آنها انجام می شود.**
تاریخی	**مقدارهای مربوط به تاریخ و ساعت**

نکته: نوع داده های جدول بالا دارای تعدادی زیرمجموعه هستند. به عنوان مثال نوع داده عددی دارای زیر مجموعه های INTEGER ، FLOAT و ... است. هر کدام از این زیرمجموعه ها دارای محدوده و فرمت نمایش متفاوت هستند. همچنین اگر از بلاک های PL/SQL در زبان های دیگر مثلا JAVA استفاده شود می بایست از نوع داده سازگار با آن زبان استفاده گردد.

زیر مجموعه های **نوع داده عددی** را در این جدول مشاهده می کنید.

نوع داده	توضیحات
PLS_INTEGER	عدد صحیح با علامت. در محدوده -2,147,483,648 تا 2,147,483,647
BINARY_INTEGER	عدد صحیح با علامت. در محدوده -2,147,483,648 تا 2,147,483,647
BINARY_FLOAT	عدد اعشاری ممیز متغیر. از نوع SINGLE-PRECISION
BINARY_DOUBLE	عدد اعشار ممیز متغیر. از نوع DOUBLE-PRECISION
NUMBER(prec, scale)	عدد اعشار ممیز متغیر یا ممیز ثابت از جمله عدد 0
DEC(prec, scale)	عدد اعشار ممیز ثابت ANSI
DECIMAL(prec, scale)	عدد اعشار ممیز ثابت IBM
NUMERIC(pre, scale)	اعداد با حداکثر دقت 38 عدد دسیمال
DOUBLE PRECISION	عدد اعشار ممیز متغیر ANSI
FLOAT	عدد اعشار ممیز متغیر ANSI و IBM
INT	اعداد از نوع ANSI با حداکثر دقت 38 عدد دسیمال
INTEGER	اعداد از نوع ANSI و IBM با حداکثر دقت 38 عدد دسیمال
SMALLINT	اعداد از نوع ANSI و IBM با حداکثر دقت 38 عدد دسیمال
REAL	اعداد اعشاری از نوع ممیز متغیر

مثال:

```
DECLARE
      num1 INTEGER;
      num2 REAL;
      num3 DOUBLE PRECISION;
BEGIN
      null;
END;
/
```

زیر مجموعه های **نوع داده کاراکتری** را در این جدول مشاهده می کنید.

نوع داده	توضیحات
CHAR	**رشته کاراکتری از نوع FIXED-LENGTH با حداکثر سایز 32767 بایت**
VARCHAR2	**رشته کاراکتری از نوع VARIABLE-LENGTH با حداکثر سایز 32767 بایت**
RAW	**رشته باینری از نوع VARIABLE-LENGTH با حداکثر سایز 32767 بایت**
NCHAR	**رشته کاراکتری NATIONAL از نوع FIXED-LENGTH با حداکثر سایز 32767 بایت**
NVARCHAR2	**رشته کاراکتری NATIONAL از نوع VARIABLE-LENGTH با حداکثر سایز 32767 بایت**
LONG	**رشته کاراکتری از نوع VARIABLE-LENGTH با حداکثر سایز 32760 بایت**
LONG RAW	**رشته باینری از نوع VARIABLE-LENGTH با حداکثر سایز 32760 بایت**
ROWID	**آدرس فیزیکی یک سطر از جدول**
UROWID	**آدرس فیزیکی یا منطقی یک سطر از جدول**

نوع داده BOOLEAN مقدارهای منطقی TRUE و FALSE و NULL را ذخیره می کند. این مقدارها می توانند در عملیات منطقی استفاده شوند.

نکته: از آنجایی که زبان SQL نوع داده BOOLEAN ندارد بنابراین مقدارهای BOOLEAN نمی توانند در دستورات یا توابع SQL استفاده شوند.

نکته: نوع داده تاریخی شامل قرن، سال، ماه، روز، ساعت و دقیقه و ثانیه می شود که می توان آنها را با فرمت مناسب نمایش داد. در دیتابیس اوراکل فرمت پیش فرض تاریخ، توسط پارامتر NLS_DATE_FORMAT تنظیم می شود.

نکته: نوع داده های LOB یا LARGE OBJECT مربوط به داده های از نوع تصویر، فیلم، صدا یا متن می شود. در جدول زیر انواع LOB های PL/SQL را می بینید.

نوع داده	توضیحات	سایز
BFILE	به منظور ذخیره داده های باینری بزرگ در فایل های سیستم عامل خارج از دیتابیس استفاده می شود.	**به سیستم بستگی دارد نهایت 4GB**
BLOB	به منظور ذخیره داده های باینری بزرگ در داخل دیتابیس استفاده می شود.	**8 – 128 TB**
CLOB	به منظور ذخیره بلاک های بزرگ کاراکتری در دیتابیس استفاده می شود.	**8 – 128 TB**
NCLOB	به منظور ذخیره بلاک های بزرگ از نوع NCHAR در دیتابیس استفاده می شود.	**8 – 128 TB**

نکته: می توان برای هرکدام از نوع داده های تعریف شده در زبان PL/SQL یک زیرمجموعه تعریف کرد که فقط محدوده خاصی از نوع داده اصلی را شامل شود.

مثال: دو نمونه از نوع داده های تعریف شده در PL/SQL که زیرمجموعه ای از نوع داده های دیگر هستند.

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

مثال: یک نوع داده 20 کاراکتری به نام name و یک نوع داده 100 کاراکتری به نام message تعریف کنید.

```
DECLARE
      SUBTYPE name IS char(20);
      SUBTYPE message IS varchar2(100);
      salutation name;
      greetings message;
BEGIN
      salutation := 'Reader ';
      greetings := 'Welcome to the World of PL/SQL';
      dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```

نکته: مقدارهای NULL در PL/SQL جزو هیچ کدام از نوع داده ها نیستند بلکه نشانگر داده ی نامفهوم یا تعریف نشده است. بنابراین مقدار NULL برابر با هیچ مقدار دیگری حتی از نوع NULL نیست.

2 متغیر ، CONSTANT و LITERAL در PL/SQL

در زبان PL/SQL متغیر (VARIABLE) نامی است که به یک محل ذخیره سازی اختصاص می یابد و برنامه ها می توانند اطلاعات خود را در این محل ذخیره کنند. هر متغیر بر اساس یک نوع داده تعریف می شود. انواع نوع داده ها را در فصل قبل توضیح دادیم. در واقع نوع داده برای یک متغیر موارد زیر را مشخص می کند.

1- سایز متغیر

2- قالب کلی متغیر

3- محدوده مقدارهایی که می توانیم در متغیر ذخیره کنیم.

4- عملیاتی که می توانیم روی آن متغیر انجام دهیم.

بنابراین به هر متغیر یک قسمت از فضای حافظه اختصاص می یابد و با استفاده از نام متغیر می توانیم به این فضا دسترسی داشته باشیم.

نکته: نام گذاری متغیرها بر اساس قوانین نام گذاری نشانگرها می باشد که در فصل قبل توضیح داده شده است. همچنین توجه شود که نمی توان از کلمات رزرو شده دیتابیس (مانند کلمه TABLE) به عنوان نام متغیر استفاده کرد.

در PL/SQL می توان متغیرها را در دو قسمت تعریف کرد:

1- درقسمت DECLARE برنامه

2- تعریف متغیر از نوع GLOBAL در یک پکیج.

سینتکس تعریف یک متغیر به این شکل است:

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

در این سینتکس variable_name نام متغیر است و datatype یک نوع داده است که می تواند از قبل تعریف شده باشد یا از نوع USER DEFINED باشد.

مثال:

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

نکته: برای نوع داده ها می توان سایز، مقیاس یا دقت خاص تعیین کنیم. به این عمل CONSTRAINED DECLARATION می گویند که باعث صرفه جویی در فضای حافظه می شود.

مثال: نمونه هایی از CONSTRAINED DECLARATION

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

2.1 مقداردهی اولیه متغیرها

زمانی که یک متغیر ایجاد می کنیم مقدار آن به صورت پیش فرض برابر با NULL است. برای اختصاص مقدار پیش فرض غیر NULL به متغیرها می توان به دو روش زیر عمل نمود:

1- استفاده از کلمه کلیدی DEFAULT

2- استفاده از عملگر مساوی :=

مثال:

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

نکته: می توانیم برای متغیرها از NOT NULL CONSTRAINT استفاده کنیم که در این صورت می بایست آن متغیر یک مقدار اولیه داشته باشد.

نکته: پیشنهاد می شود به تمام متغیرها یک مقدار اولیه مناسب اختصاص داده شود. در غیر این صورت ممکن است برنامه های PL/SQL نتایج نامناسب تولید کنند.

مثال: کاربرد متغیرها و نمایش خروجی آنها

```
DECLARE
        a integer := 10;
        b integer := 20;
        c integer;
        f real;
BEGIN
        c := a + b;
        dbms_output.put_line('Value of c: ' || c);
        f := 70.0/3.0;
        dbms_output.put_line('Value of f: ' || f);
END;
/
```

خروجی برنامه:

Value of c: 30
Value of f: 23.333333333333333
PL/SQL procedure successfully completed.

2.2 محدوده عملیاتی متغیرها

همانطور که می دانیم می توان در داخل یک بلاک برنامه PL/SQL از بلاک های دیگر استفاده نمود. در این حالت متغیری که در بلاک بیرونی تعریف شده است برای تمام بلاک های درونی قابل دسترس است ولی متغیرهای بلاک های درونی برای بلاک های بیرونی قابل دسترس نیستند. بنابراین متغیرها دو دسته هستند:

1- LOCAL: متغیرهایی که در یک بلاک درونی تعریف شده اند و برای بلاک بیرونی قابل دسترس نیستند.

2- GLOBAL: متغیرهایی که در بلاک بیرونی تعریف شده اند یا در یک پکیج تعریف می شوند و برای تمام بلاک های درونی قابل دسترس هستند.

مثال:

```
DECLARE
        -- Global variables
        num1 number := 95;
        num2 number := 85;
BEGIN
        dbms_output.put_line('Outer Variable num1: ' || num1);
        dbms_output.put_line('Outer Variable num2: ' || num2);
        DECLARE
                -- Local variables
                num1 number := 195;
                num2 number := 185;
        BEGIN
                dbms_output.put_line('Inner Variable num1: ' || num1);
                dbms_output.put_line('Inner Variable num2: ' || num2);
        END;
END;
/
```

خروجی برنامه:

Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185
PL/SQL procedure successfully completed

2.3 اختصاص نتایج دستورات SQL به متغیرهای PL/SQL

برای اختصاص مقدارهای SQL به متغیرهای PL/SQL از دستور SELECT INTO استفاده می شود. در این دستور باید برای هر مقدار در لیست SELECT یک متغیر با نوع داده مناسب و سازگار در لیست INTO وجود داشته باشد.

مثال: ایجاد یک جدول:

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL,
ADDRESS CHAR (25),
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);
Table Created
```

درج مقدار در جدول:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

برنامه PL/SQL که از مقدارهای جدول بالا استفاده می کند:

عبارت %type به معنی نوع داده برابر با ستون جدول اشاره شده است.

```
DECLARE
    c_id customers.id%type := 1;
    c_name customers.name%type;
    c_addr customers.address%type;
    c_sal customers.salary%type;
BEGIN
    SELECT name, address, salary INTO c_name, c_addr, c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line
    ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

خروجی برنامه:

```
Customer Ramesh from Ahmedabad earns 2000
PL/SQL procedure completed successfully
```

CONSTANT 2.4

CONSTANT مقداری ثابت است که قابل تغییر نیست. برای تعریف یک CONSTANT همانند متغیرها باید از نام و نوع داده مناسب استفاده کنیم. همچنین می توان NOT NULL CONSTRAINT برای آنها درنظر گرفت.

برای تعریف CONSTANT از کلمه کلیدی CONSTANT استفاده می شود.

مثال:

```
DECLARE
        -- constant declaration
        pi constant number := 3.141592654;
        -- other declarations
        radius number(5,2);
        dia number(5,2);
        circumference number(7, 2);
        area number (10, 2);
BEGIN

        -- processing

        radius := 9.5;
        dia := radius * 2;
        circumference := 2.0 * pi * radius;
        area := pi * radius * radius;
        -- output
        dbms_output.put_line('Radius: ' || radius);
        dbms_output.put_line('Diameter: ' || dia);
        dbms_output.put_line('Circumference: ' || circumference);
        dbms_output.put_line('Area: ' || area);
END;
/
```

خروجی برنامه:

Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53
Pl/SQL procedure successfully completed.

2.5 LITERAL چیست؟

LITERAL یک مقدار از نوع کاراکتری، عددی یا BOOLEAN است که به صورت صریح تعریف می شود و نام یا نشانگر ندارد. برای مثال TRUE، 786 ، NULL و 'fortest' همه LITERALهایی هستند که از نوع داده های مختلف هستند.

نکته: LITERALها حساس به بزرگی و کوچکی حروف هستند.

مثال هایی از LITERAL برای نوع داده های مختلف را جدول زیر می بینید

LITERALنوع	مثال
عددی	050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
کاراکتری	'A' '%' '9' ' ' 'z' '('
رشته ای	'Hello, world!' 'Tutorials Point' '19-NOV-12'
BOOLEAN	TRUE, FALSE, and NULL
تاریخی و عددی	DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01';

مثال:

```
DECLARE
        message varchar2(30):= 'That''s tutorialspoint.com!';
BEGIN
        dbms_output.put_line(message);
END;
/
```

خروجی مثال بالا:

```
That's tutorialspoint.com!
PL/SQL procedure successfully completed.
```

نکته: در مثال بالا استفاده از دو علامت ' پشت سر هم باعث می شود این علامت به عنوان دلیمتر در نظر گرفته نشود و نادیده گرفته شود.

3 عملگرهای PL/SQL

کامپایلر زبان PL/SQL با توجه به نوع عملگر استفاده شده در برنامه، عملیات خاصی را برروی داده ها انجام می دهد. عملگرها به 5 دسته زیر تقسیم می شوند:

- عملگرهای ریاضی
- عملگرهای رابطه ای
- عملگرهای مقایسه ای
- عملگرهای منطقی
- عملگرهای رشته ای

در این فصل هر کدام از ان عملگرها را با مثال توضیح می دهیم.

3.1 عملگرهای ریاضی

در جدول زیر متغیر A برابر با 10 و متغیر B برابر 5 است.

عملگر	توضیحات	مثال
+	دو عملوند را جمع می کند.	**A + B => 15**
–	دومین عملوند را از عملوند اول کم می کند.	**A – B => 5**
*	عمل ضرب را انجام می دهد.	**A * B => 50**
/	عمل تقسیم را انجام می دهد.	**A / B => 2**
**	عدد اول را به توان عدد دوم می رساند.	**A ** B => 100000**

مثال:

```
BEGIN
        dbms_output.put_line( 10 + 5);
        dbms_output.put_line( 10 - 5);
        dbms_output.put_line( 10 * 5);
        dbms_output.put_line( 10 / 5);

        dbms_output.put_line( 10 ** 5);

END;
/
```

خروجی

```
15
5
50
2
100000
PL/SQL procedure successfully completed.
```

3.2 عملگرهای رابطه ای

عملگرهای رابطه ای، دو عبارت را مقایسه می کنند و یک مقدار BOOLEAN (مقدار TRUE یا FALSE) برمی گردانند. در جدول زیر عملگرهای رابطه ای را می بینید. فرض کنید عملوند A برابر 10 و عملوند B برابر با 20 است.

عملگر	توضیحات	مثال
=	بررسی مساوی بودن	(A = B) => FALSE
=~ یا <>یا !=	نا مساوی بودن را بررسی می کند	(A != B)=> TRUE
>	بزرگتر از	(A > B) => FALSE

<	کوچکتر از	(A< B) => TRUE
>=	بزرگتر یا مساوی	(A >= B) => TRUE
<=	کوچکتر یا مساوی	(A <= B) => TRUE

مثال:

```
DECLARE
        a number (2) := 21;
        b number (2) := 10;
BEGIN
        IF (a = b) then
                dbms_output.put_line('Line 1 - a is equal to b');
        ELSE
                dbms_output.put_line('Line 1 - a is not equal to b');
        END IF;
        IF (a < b) then
                dbms_output.put_line('Line 2 - a is less than b');
        ELSE
                dbms_output.put_line('Line 2 - a is not less than b');
        END IF;
        IF ( a > b ) THEN
                dbms_output.put_line('Line 3 - a is greater than b');
        ELSE
                dbms_output.put_line('Line 3 - a is not greater than b');
        END IF;
        -- Lets change value of a and b
        a := 5;
        b := 20;
        IF ( a <= b ) THEN
                dbms_output.put_line('Line 4 - a is either equal or less than b');
        END IF;
        IF ( b >= a ) THEN
                dbms_output.put_line('Line 5 - b is either equal or greater than a');
        END IF;

        IF ( a <> b ) THEN
```

```
            dbms_output.put_line('Line 6 - a is not equal to b');
        ELSE
            dbms_output.put_line('Line 6 - a is equal to b');
        END IF;
END;
```

خروجی

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either equal or less than b
Line 5 - b is either equal or greater than a
Line 6 - a is not equal to b
PL/SQL procedure successfully completed
```

3.3 عملگرهای مقایسه ای

وقتی از عملگرهای مقایسه ای استفاده می شود نتیجه TRUE ، FALSE یا NULL است.

عملگر	توضیحات
LIKE	**وجود یک الگو را بررسی می کند**
BETWEEN	**بررسی اینکه مقدار در یک محدوده خاص است یا نه**
IN	**عضویت در یک مجموعه را بررسی می کند.**
IS NULL	**بررسی NULL بودن عملوند**

مثال از LIKE

```
DECLARE
        PROCEDURE compare (value varchar2, pattern varchar2 ) is
        BEGIN
                IF value LIKE pattern THEN
                        dbms_output.put_line ('True');
                ELSE
                        dbms_output.put_line ('False');
                END IF;
        END;
BEGIN
        compare('Zara Ali', 'Z%A_i');
        compare('Nuha Ali', 'Z%A_i');
END;
/
```

خروجی برنامه:

```
True
False
PL/SQL procedure successfully completed.
```

مثال از BETWEEN

```
DECLARE
        x number(2) := 10;
BEGIN
        IF (x between 5 and 20) THEN
                dbms_output.put_line('True');
        ELSE
                dbms_output.put_line('False');
        END IF;
        IF (x BETWEEN 5 AND 10) THEN
                dbms_output.put_line('True');
        ELSE
                dbms_output.put_line('False');
        END IF;
        IF (x BETWEEN 11 AND 20) THEN
```

```
            dbms_output.put_line('True');
        ELSE
            dbms_output.put_line('False');
        END IF;
END;
/
```

خروجی برنامه:

```
True
True
False
PL/SQL procedure successfully completed.
```

مثال از IN و IS NULL

```
DECLARE
        letter varchar2(1) := 'm';
BEGIN
        IF (letter in ('a', 'b', 'c')) THEN
            dbms_output.put_line('True');
        ELSE
            dbms_output.put_line('False');
        END IF;
        IF (letter in ('m', 'n', 'o')) THEN
            dbms_output.put_line('True');
        ELSE
            dbms_output.put_line('False');
        END IF;
        IF (letter is null) THEN
            dbms_output.put_line('True');
        ELSE
            dbms_output.put_line('False');
        END IF;
END;
/
```

خروجی برنامه:

False
True
False
PL/SQL procedure successfully completed.

3.4 عملگرهای منطقی

در جدول زیر عملگرهای منطقی را می بینید. این عملگرها روی مقدارهای BOOLEAN عمل می کنند و نتیجه آنها نیز TRUE یا FALSE است. فرض کنید متغیر A برابر با TRUE و B برابر با FALSE است.

عملگر	توضیحات	مثال
AND	اگر هر دو عملوند TRUE باشد TRUE است در غیر این صورت FALSE است	**(A AND B) => FALSE**
OR	اگر یکی از عملوند ها TRUE باشد TRUE و اگر هر دو FALSE باشد FALSE است.	**(A OR B) => TRUE**
NOT	عملوند را از لحاظ منطقی معکوس می کند	**NOT (A AND B) => TRUE**

مثال:

```
DECLARE
        a boolean := true;
        b boolean := false;
BEGIN
        IF (a AND b) THEN
                dbms_output.put_line('Line 1 - Condition is true');
        END IF;
        IF (a OR b) THEN
```

```
        dbms_output.put_line('Line 2 - Condition is true');
    END IF;
    IF (NOT a) THEN
        dbms_output.put_line('Line 3 - a is not true');
    ELSE
        dbms_output.put_line('Line 3 - a is true');
    END IF;
    IF (NOT b) THEN
        dbms_output.put_line('Line 4 - b is not true');
    ELSE
        dbms_output.put_line('Line 4 - b is true');
    END IF

END;
/
```

خروجی برنامه:

```
Line 2 - Condition is true
Line 3 - a is true
Line 4 - b is not true
PL/SQL procedure successfully completed.
```

3.5 اولویت عملگرها

در جدول زیر عملگرهای با اولویت بالاتر در سطر بالا قرار دارند. برای مثال اگر در یک برنامه x=7+3*2 باشد مقدار x برابر با 13 است زیرا عملگر ضرب (سطر سوم) به نسبت جمع(سطر چهارم) اولویت بالاتر دارد.

توضیحات	عملگر
عدد اول به توان عدد دوم	**
مثبت یا منفی بودن عدد	+, -
ضرب و تقسیم	*, /
جمع، تفریق و الحاق	+, -, ||
مقایسه گر ها	عملگرهای مقایسه ای

NOT	عملگر منطقی
AND	عملگر منطقی
OR	عملگر منطقی

مثال:

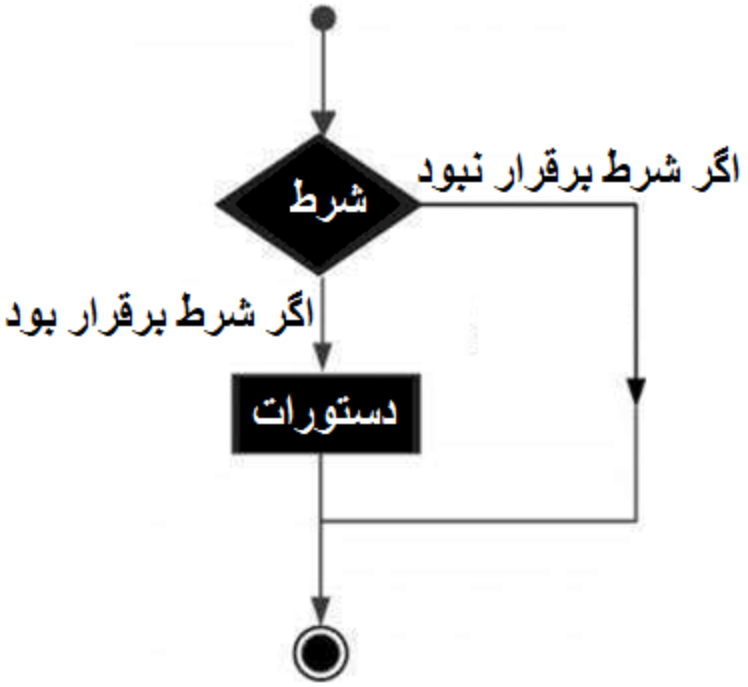
```
DECLARE
        a number(2) := 20;
        b number(2) := 10;
        c number(2) := 15;
        d number(2) := 5;
        e number(2) ;
BEGIN
        e := (a + b) * c / d; -- ( 30 * 15 ) / 5
        dbms_output.put_line('Value of (a + b) * c / d is : '|| e );
        e := ((a + b) * c) / d; -- (30 * 15 ) / 5
        dbms_output.put_line('Value of ((a + b) * c) / d is : ' || e );
        e := (a + b) * (c / d); -- (30) * (15/5)
        dbms_output.put_line('Value of (a + b) * (c / d) is : '|| e );
        e := a + (b * c) / d; -- 20 + (150/5)
        dbms_output.put_line('Value of a + (b * c) / d is : ' || e );
END;
/
```

خروجی برنامه:

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is : 90
Value of (a + b) * (c / d) is : 90
Value of a + (b * c) / d is : 50
PL/SQL procedure successfully completed.
```

4 دستورات شرطی

گاهی اوقات نیاز است یک یا چند شرط بررسی شوند تا عملیات برنامه بر اساس آن شرط ها انجام گیرند. در شکل زیر ساختار کلی دستورات شرطی را مشاهده می کنید که در اکثر زبان های برنامه نویسی به همین ترتیب است. این دستورات برای تغییر روند کنترلی اجرای دستورات برنامه استفاده می شود.



در جدول زیر انواع دستورات شرطی PL/SQL و توضیحات آنها را ملاحظه می کنید.

دستور شرطی	توضیحات
IF - THEN	**در این دستور شرطی اگر شرط IF برقرار باشد دستورات بعد از کلمه کلیدی THEN اجرا می شود. اگر شرط برقرار نباشد یا NULL باشد هیچ دستوری اجرا نخواهد یافت. در هر حالت کلمه کلیدی END IF به معنی پایان دستور شرطی IF - THEN است.**
IF THEN - ELSE	**همانند دستور شرطی IF – THEN است ولی زمانی که شرط برقرار نیست یا NULL است دستورات بعد از کلمه کلیدی ELSE اجرا خواهند شد.**
IF – THEN - ELSIF	**همانند دستور شرطی IF – THEN – ELSE است با این تفاوت که می توانیم دستورات را بر اساس چندین شرط متفاوت دسته بندی کنیم.**
CASE	**همانند دستورات شرطی IF عمل می کند با این تفاوت که بعد از کلمه کلیدی CASE از یک انتخابگر استفاده می شود و دستورات متفاوت، براساس مقدار آن انتخابگر دسته بندی می شوند.**
Searched CASE	**در این دستور بعد از کلمه کلیدی CASE از انتخابگر استفاده نمی شود بلکه هر عبارت WHEN شامل یک منطق شرطی است.**
IF – THEN – ELSE تودرتو	**دستورات شرطی IF – THEN یا IF – THEN – ELSE می توانند به صورت تودرتو استفاده شوند.**

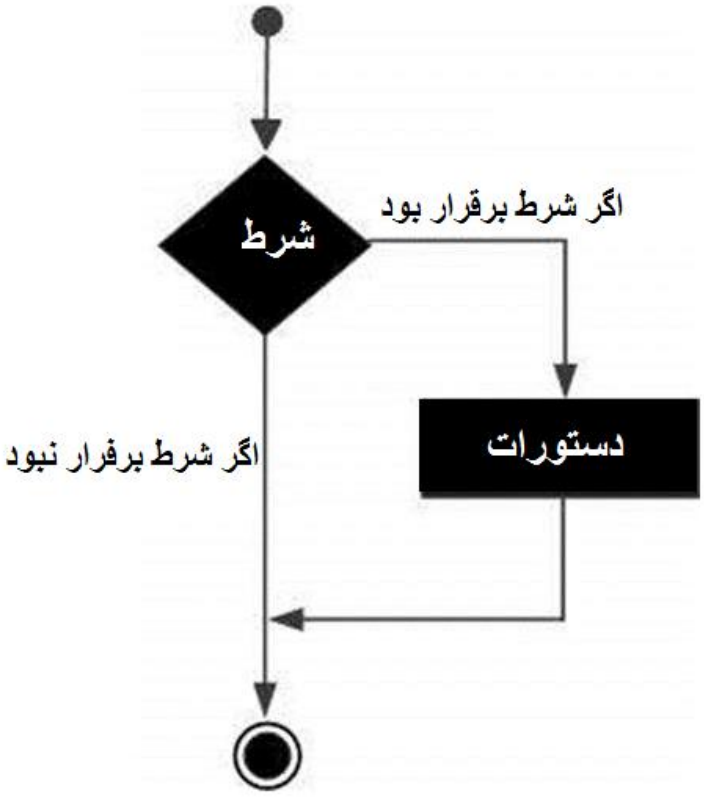
4.1 دستور شرطی IF – THEN

ساده ترین دستور شرطی دستور IF – THEN است. سینتکس این دستور به این شکل است:

```
IF condition THEN
        Statement;
END IF;
```

در این سینتکس CONDITION شرط دستور و Statement بلاک دستورات است.

دیاگرام دستور شرطی IF – THEN را در شکل زیر می بینید.



مثال:

```
IF (a <= 20) THEN
        c:= c+1;
END IF;
```

مثال:

```
DECLARE
        a number(2) := 10;
BEGIN
        a:= 10;
        -- check the boolean condition using if statement
        IF( a < 20 ) THEN
        -- if condition is true then print the following
        dbms_output.put_line('a is less than 20 ' );
        END IF;
        dbms_output.put_line('value of a is : ' || a);
END;
/
```

خروجی:

```
a is less than 20
value of a is : 10
PL/SQL procedure successfully completed.
```

مثال:

```
DECLARE
        c_id customers.id%type := 1;
        c_sal customers.salary%type;
BEGIN
        SELECT salary
        INTO c_sal
        FROM customers
        WHERE id = c_id;
        IF (c_sal <= 2000) THEN
                UPDATE customers
                SET salary = salary + 1000
                WHERE id = c_id;
```

```
            dbms_output.put_line ('Salary updated');
        END IF;
END;
/
```

خروجی:

```
Salary updated
PL/SQL procedure successfully completed.
```

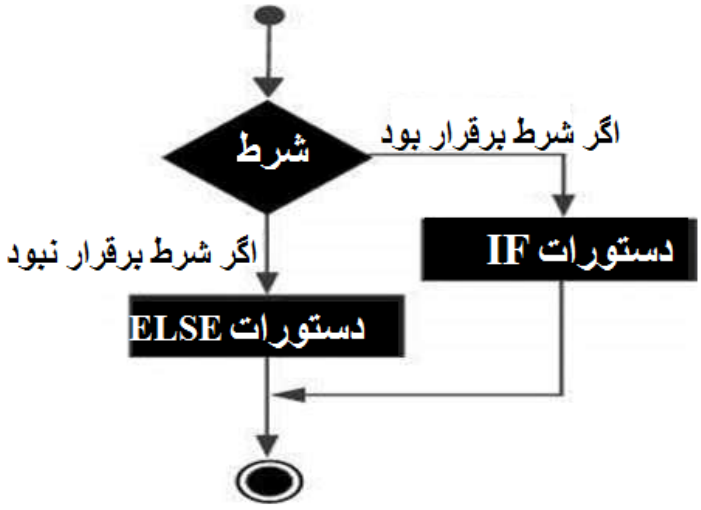
4.2 دستور شرطی IF – THEN – ELSE

سینتکس این دستور شرطی به این شکل است:

```
IF condition THEN
        Statement1;
ELSE
        Statement2;
END IF;
```

در این سینتکس CONDITION شرط دستور و Statement1 و Statemenet2 بلاک دستورات هستند.

دیاگرام این دستور را در شکل زیر می بینید:



مثال:

```
DECLARE
        a number(3) := 100;
BEGIN
        -- check the boolean condition using if statement
        IF( a < 20 ) THEN
        -- if condition is true then print the following
                dbms_output.put_line('a is less than 20 ' );
        ELSE
                dbms_output.put_line('a is not less than 20 ' );
        END IF;
        dbms_output.put_line('value of a is : ' || a);
END;
/
```

خروجی:

```
a is not less than 20
value of a is : 100
PL/SQL procedure successfully completed.
```

4.3 دستور شرطی IF – THEN – ELSIF

سینتکس این دستور شرطی به این شکل است:

```
IF(boolean_expression 1)THEN
        Statement1;
ELSIF( boolean_expression 2) THEN
        Statement2;
ELSIF( boolean_expression 3) THEN
        Statement3;
ELSE
        Sstatement4;
END IF;
```

نکته: هرکدام از ELSIF ها بر قرار باشند ELSIFهای باقی مانده بررسی نمی شوند.

نکته: در هر دستور شرطی IF – THEN می توان صفر یا چندین ELSIF داشت ولی فقط می توان صفر یا یک عبارت ELSE داشت که بعد از تمامی ELSIF ها نوشته می شود.

مثال:

```
DECLARE
        a number(3) := 100;
BEGIN
        IF ( a = 10 ) THEN

                dbms_output.put_line('Value of a is 10' );
        ELSIF ( a = 20 ) THEN
                dbms_output.put_line('Value of a is 20' );
        ELSIF ( a = 30 ) THEN
                dbms_output.put_line('Value of a is 30' );
        ELSE
                dbms_output.put_line('None of the values is matching');
        END IF;
        dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

خروجی:

```
None of the values is matching
Exact value of a is: 100
PL/SQL procedure successfully completed.
```

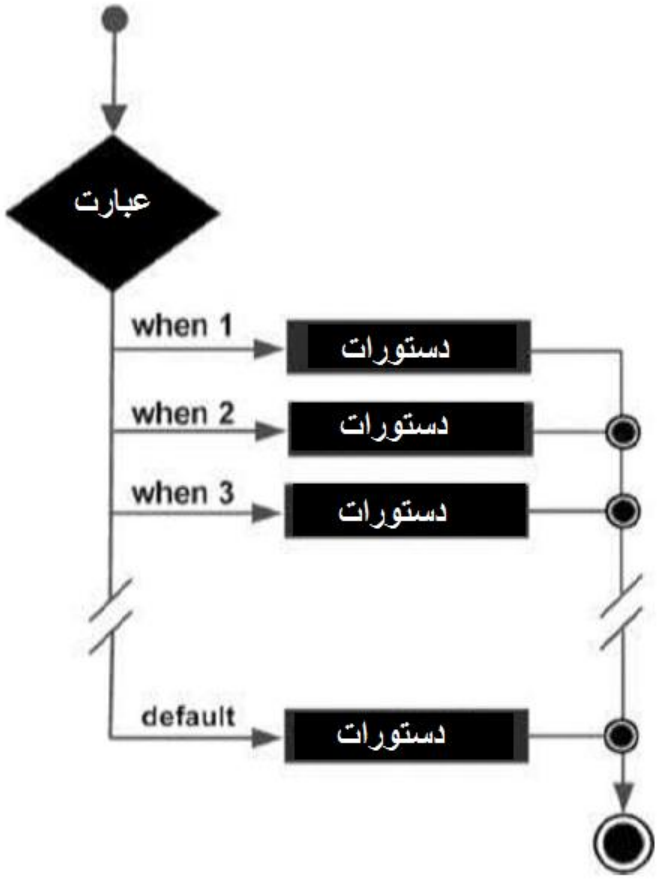
4.4 دستور شرطی CASE

سینتکس دستور شرطی CASE به این شکل است:

```
CASE selector
        WHEN 'value1' THEN Statement1;
        WHEN 'value2' THEN Statement2;
        WHEN 'value3' THEN Statement3;
        ...
        ELSE Statementn;
END CASE;
```

در این سینتکس selector انتخابگر دستور است که مقدار آن در عبارت های WHEN بررسی می شود و تا دستورات (statement) مناسب اجرا شود.

دیاگرام این دستور را در شکل زیر می بینید:



مثال:

```
DECLARE
        grade char(1) := 'A';
BEGIN
        CASE grade
        when 'A' then dbms_output.put_line('Excellent');
        when 'B' then dbms_output.put_line('Very good');
        when 'C' then dbms_output.put_line('Well done');
        when 'D' then dbms_output.put_line('You passed');
        when 'F' then dbms_output.put_line('Better try again');
        else dbms_output.put_line('No such grade');
        END CASE;
END;
/
```

خروجی:

```
Excellent
PL/SQL procedure successfully completed.
```

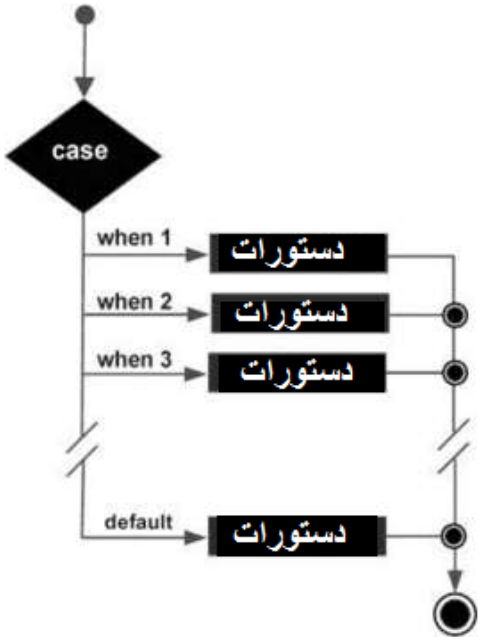
4.5 دستور شرطی SEARCHED CASE

سینتکس دستور شرطی SEARCHED CASE به این شکل است:

```
 CASE
        WHEN selector = 'value1' THEN Statement1;
        WHEN selector = 'value2' THEN Statement2;
        WHEN selector = 'value3' THEN Statement3;
        ...
        ELSE Statemenetn; -- default case
END CASE;
```

همانطور که مشاهده می کنید شرط در عبارت WHEN تعیین می شود.

دیاگرام این دستور را در شکل زیر می بینید:



مثال:

```
DECLARE
        grade char(1) := 'B';
BEGIN
        case
        when grade = 'A' then dbms_output.put_line('Excellent');
        when grade = 'B' then dbms_output.put_line('Very good');
        when grade = 'C' then dbms_output.put_line('Well done');
        when grade = 'D' then dbms_output.put_line('You passed');
        when grade = 'F' then dbms_output.put_line('Better try again');
        else dbms_output.put_line('No such grade');
        end case;
END;
/
```

خروجی:

```
Very good
PL/SQL procedure successfully completed.
```

4.6 IF – THEN – ELSE تودرتو

سینتکس **IF – THEN – ELSE** تودرتو به این شکل است:

```
IF (boolean_expression 1)THEN
        IF(boolean_expression 2) THEN
                sequence-of-statements;
        END IF;
ELSE
        else-statements;
END IF;
```

مثال:

```
DECLARE
        a number(3) := 100;
        b number(3) := 200;
BEGIN
        -- check the boolean condition
        IF( a = 100 ) THEN
        -- if condition is true then check the following
                IF( b = 200 ) THEN
                -- if condition is true then print the following
                        dbms_output.put_line('Value of a is 100 and b is 200' );
                END IF;
        END IF;
        dbms_output.put_line('Exact value of a is : ' || a );
        dbms_output.put_line('Exact value of b is : ' || b );
END;
/
```

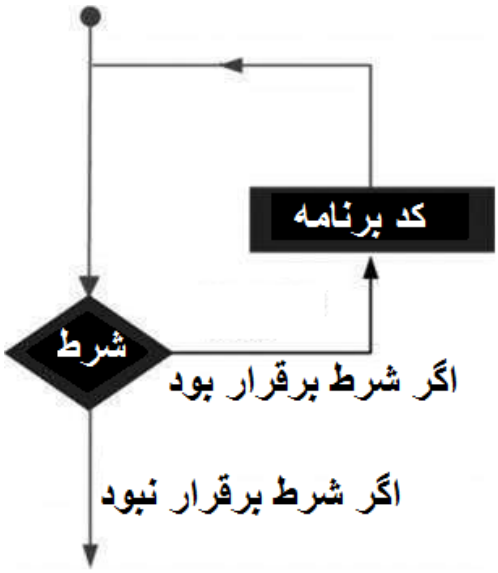
خروجی:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
PL/SQL procedure successfully completed.
```

5 انواع حلقه در PL/SQL

در این فصل انواع LOOP یا حلقه در PL/SQL را بررسی می کنیم. ممکن است در یک برنامه نیاز باشد یک بلاک از کد، چندین مرتبه اجرا شود در این مواقع از حلقه استفاده می گردد.

در شکل زیر دیاگرام اجرای دستورات حلقه را می بینید که در اکثر زبان های برنامه نویسی به همین روال است.



در جدول زیر انواع روش های حلقه و توضیح آنها را مشاهده می کنید.

روش حلقه	توضیحات
حلقه ساده	**در این روش دستورات بین کلمه کلیدی LOOP و END LOOP قرار دارند و در هر مرحله از اجرای حلقه این دستورات به صورت ترتیبی اجرا می شوند و سپس کنترل برنامه به ابتدای حلقه بازمی گردد**
حلقه WHILE	**در حلقه WHILE تا زمانی که یک شرط برقرار باشد اجرای دستورات داخلی ادامه می یابد. این شرط قبل از هر مرحله از اجرای دستورات داخلی دوباره بررسی می شود.**
حلقه FOR	**همانند روش WHILE است ولی متغیر کنترل حلقه در دستور FOR بررسی می شود.**
حلقه های تودرتو	**می توان هر تعداد حلقه در داخل انواع روش های حلقه استفاده نمود.**

5.1 حلقه ساده

سینتکس حلقه ساده به این شکل است:

```
LOOP
        Sequence of statements;
END LOOP;
```

در این سینتکس Sequence of statements می تواند یک دستور یا یک بلاک از دستورات باشد.

نکته: برای خروج از حلقه ساده باید از دستور EXIT یا EXIT WHEN استفاده نمود.

مثال:

```
DECLARE
        x number := 10;
BEGIN
        LOOP
                dbms_output.put_line(x);
                x := x + 10;
                IF x > 50 THEN
                        exit;
                END IF;
        END LOOP;
        -- after exit, control resumes here
        dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

خروجی

```
10
20
30
40
50
After Exit x is: 60
PL/SQL procedure successfully completed.
```

مثال: برای خروج از حلقه از دستور EXIT WHEN استفاده شده است.

```
DECLARE
        x number := 10;
BEGIN
        LOOP
```

```
            dbms_output.put_line(x);
            x := x + 10;
            exit WHEN x > 50;
        END LOOP;
        -- after exit, control resumes here
        dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

خروجی

```
10
20
30
40
50
After Exit x is: 60
PL/SQL procedure successfully completed.
```

5.2 حلقه WHILE

سینتکس حلقه WHILE به این شکل است:

```
WHILE condition LOOP
        sequence_of_statements
END LOOP;
```

مثال:

```
DECLARE
        a number(2) := 10;
BEGIN
```

```
        WHILE a < 20 LOOP
                dbms_output.put_line('value of a: ' || a);
                a := a + 1;
        END LOOP;
END;
/
```

خروجی

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
PL/SQL procedure successfully completed.
```

5.3 سینتکس حلقه FOR

```
FOR counter IN initial_value .. final_value LOOP
        sequence_of_statements;
END LOOP;
```

نکته: در این روش initial value و final value می توانند از نوع LITERAL ، متغیر یا یک عبارت باشند به شرطی که به عنوان اعداد باشند وگرنه خطای VALUE_ERROR دریافت می کنیم.

نکته: نیازی نیست که intial value در یک حلقه FOR برابر با یک باشد ولی افزایش و کاهش شمارنده باید یک باشد.

نکته: می توان محدوده initial value و final value را به صورت دینامیک و در زمان اجرا مشخص نمود.

مثال:

```
DECLARE
        a number(2);
BEGIN
        FOR a in 10 .. 20 LOOP
                dbms_output.put_line('value of a: ' || a);
        END LOOP;
END;
/
```

خروجی

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
PL/SQL procedure successfully completed.
```

نکته: زمانی که می خواهیم در حلقه FOR به صورت معکوس شمارش انجام شود از کلمه کلیدی REVERSE استفاده می گردد. در ضمن باید در دستور FOR محدوده شمارش را به همان صورت صعودی مشخص کرد.

مثال:

```
DECLARE
        a number(2) ;
BEGIN
        FOR a IN REVERSE 10 .. 20 LOOP
                dbms_output.put_line('value of a: ' || a);
        END LOOP;
END;
/
```

خروجی

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10
PL/SQL procedure successfully completed.
```

5.4 حلقه های تودرتو

در ادامه سینتکس های مربوط به روش های حلقه تودرتو را می بینید:

```
LOOP
        Sequence of statements1
        LOOP
                Sequence of statements2
        END LOOP;
END LOOP;
```

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
        sequence_of_statements1
        FOR counter2 IN initial_value2 .. final_value2 LOOP
                sequence_of_statements2
        END LOOP;
END LOOP;
```

```
WHILE condition1 LOOP
        sequence_of_statements1
        WHILE condition2 LOOP
                sequence_of_statements2
        END LOOP;
END LOOP;
```

مثال: برنامه ای بنویسید که اعداد اول بین 2 تا 50 را نمایش دهد.

```
DECLARE
        i number(3);
        j number(3);
BEGIN
        i := 2;
        LOOP
                j:= 2;
                LOOP
```

```
                exit WHEN ((mod(i, j) = 0) or (j = i));
                j := j +1;
            END LOOP;
        IF (j = i ) THEN
            dbms_output.put_line(i || ' is prime');
        END IF;
        i := i + 1;
        exit WHEN i = 50;
        END LOOP;
END;
/
```

خروجی

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
PL/SQL procedure successfully completed.
```

نکته: با علامت براکت << >> می توان یک LABEL یا برچسب برای حلقه در نظرگرفت که قبل از دستور حلقه قرار می گیرد.

مثال:

```
DECLARE
        i number(1);
        j number(1);
BEGIN
        << outer_loop >>
        FOR i IN 1..3 LOOP
                << inner_loop >>
                FOR j IN 1..3 LOOP
                        dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
                END loop inner_loop;
        END loop outer_loop;
END;
/
```

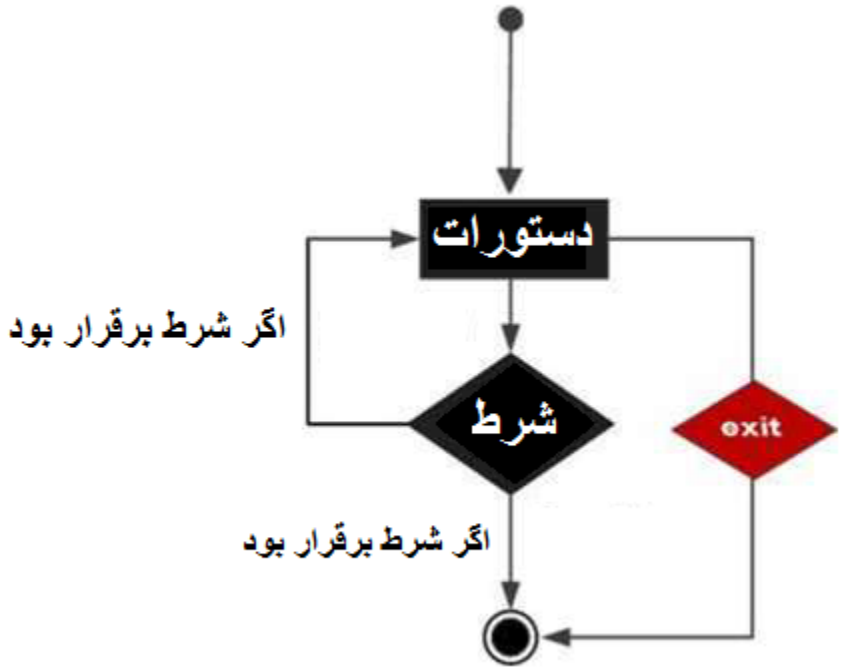
خروجی

```
i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3
PL/SQL procedure successfully completed.
```

5.5 دستورات کنترلی حلقه ها

دستور کنترلی	توضیحات
دستور EXIT	**هر زمان این دستور استفاده می شود اجرای حلقه متوقف می گردد و کنترل اجرای برنامه به اولین دستور بعد از حلقه منتقل می شود.**
دستور CONTINUE	**سبب می شود ادامه دستورات حلقه نادیده گرفته شود و مجددا شرط حلقه بررسی گردد.**
دستور GOTO	**می توان کنترل برنامه را به یک دستور دارای LABEL یا برچسب منتقل نمود هرچند استفاده از این دستور توصیه نمی شود.**

دیاگرام دستور EXIT را در شکل زیر می بینید.



مثال:

```
DECLARE
        a number(2) := 10;
BEGIN
        -- while loop execution
        WHILE a < 20 LOOP
                dbms_output.put_line ('value of a: ' || a);
                a := a + 1;
                IF a > 15 THEN
                -- terminate the loop using the exit statement
                        EXIT;
                END IF;
        END LOOP;
END;
/
```

خروجی

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
PL/SQL procedure successfully completed.
```

نکته: اگر از دستور EXIT WHEN استفاده شود یک شرط برای خارج شدن از حلقه بررسی می گردد بنابراین استفاده از این دستور مانند زمانی است که از IF-THEN استفاده شود. اگر شرط برقرار نباشد این دستور مانند دستور NULL عمل می کند.

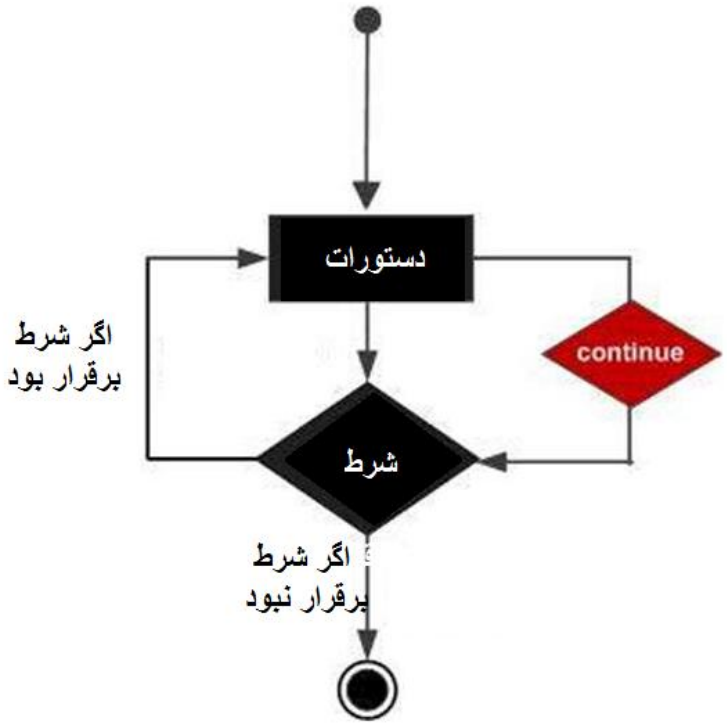
مثال:

```
DECLARE
        a number(2) := 10;
BEGIN
        -- while loop execution
        WHILE a < 20 LOOP
                dbms_output.put_line ('value of a: ' || a);
                a := a + 1;
                -- terminate the loop using the exit when statement
                EXIT WHEN a > 15;
        END LOOP;
END;
/
```

خروجی

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
PL/SQL procedure successfully completed.
```

در شکل زیر دیاگرام دستور CONTINUE را می بینید.



مثال:

```
DECLARE
        a number(2) := 10;
BEGIN
        -- while loop execution
        WHILE a < 20 LOOP
                dbms_output.put_line ('value of a: ' || a);
                a := a + 1;
                IF a = 15 THEN
                        -- skip the loop using the CONTINUE statement
                        a := a + 1;
                        CONTINUE;
                END IF;
        END LOOP;
```

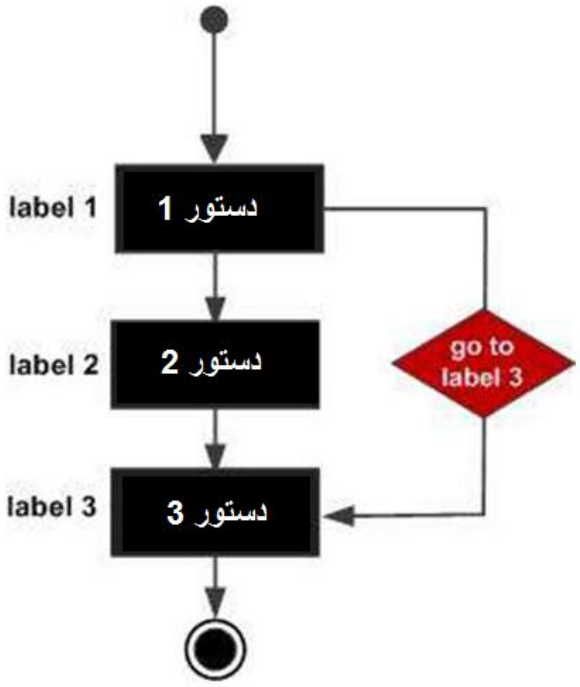
END;
/

خروجی

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
PL/SQL procedure successfully completed.

نکته: از آنجایی که دستور GOTO سبب پیچیدگی در فهم برنامه می شود استفاده از آن توصیه نمی گردد.

در شکل زیر دیاگرام دستور GOTO را می بینید.



مثال:

```
DECLARE
        a number(2) := 10;
BEGIN
        <<loopstart>>
        -- while loop execution
        WHILE a < 20 LOOP
                dbms_output.put_line ('value of a: ' || a);
                a := a + 1;
                IF a = 15 THEN
                        a := a + 1;
                        GOTO loopstart;
                END IF;
        END LOOP;
END;
/
```

خروجی

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
PL/SQL procedure successfully completed.
```

نکته: دستور GOTO فقط در بلاک فعلی عمل می کند و نمی توان از طریق آن به یک بلاک درونی انتقال یافت.

نکته: در داخل یک EXCEPTION HANDLER نمی توان از GOTO برای بازگشت به بلاک فعلی استفاده نمود.

نکته: نمی توان با دستور GOTO به داخل یک IF ، CASE یا حلقه انتقال یافت.

6 رشته و آرایه در PL/SQL

6.1 رشته

رشته (STRING)، تعداد مشخصی از داده های کاراکتری است. کاراکترهای رشته می توانند از نوع عددی، حروف، جای خالی یا ترکیبی از آنها باشند. در PL/SQL سه نوع رشته داریم:

1- رشته های از نوع FIXED-LENGTH:

در این نوع رشته ها فضای مورد استفاده برابر با حداکثر طول تعیین شده خواهد بود. مانند نوع داده CHAR

مثال: در این دو رشته حداکثر طول برابر یک است.

```
red_flag CHAR(1) := 'Y';
red_flag CHAR := 'Y';
```

مثال: فضای حافظه رشته زیر معادل با 10 کاراکتر است.

```
TEST CHAR(10) := 'HELLO';
```

2- رشته های از نوع VARIABLE-LENGTH:

در این نوع رشته ها فضای مورد استفاده برابر با حداکثر طول رشته **وارد شده** خواهد بود. البته باید حداکثر طول داده را مشخص نمود. مانند نوع داده VARCHAR2.

مثال: فضای حافظه رشته زیر معادل با 5 کاراکتر است.

```
TEST CHAR(10) := 'HELLO';
```

مثال: رشته name برابر با 'John Smith' است پس فضای مورد نیاز برابر با 10 کاراکتر خواهد بود زیرا رشته name از نوع varchar2 است.

```
DECLARE
      name varchar2(20);
```

```
      company varchar2(30);
      introduction clob;
      choice char(1);
BEGIN
      name := 'John Smith';
      company := 'Infotech';
      introduction := ' Hello! I''m John Smith from Infotech.';

      choice := 'y';
      IF choice = 'y' THEN
            dbms_output.put_line(name);
            dbms_output.put_line(company);
            dbms_output.put_line(introduction);
      END IF;
END;
/
```

خروجی

```
John Smith
Infotech Corporation
Hello! I'm John Smith from Infotech.
PL/SQL procedure successfully completed
```

3- رشته های از نوع CLOB(Character Large Object):

این رشته ها از نوع VARIABLE-LENGTH هستند که طول داده می تواند تا 128TB باشد.

نکته: نوع داده های رشته هایی که با N شروع می شوند از نوع NATINAL CHARACTER SET هستند مانند NVARCHAR2.

نکته: در PL/SQL رشته ها می توانند از نوع متغیر یا LITERAL باشد.

مثال: تعریف رشته به صورت LITERAL.

```
'This is a string literal.'
'hello world'
```

6.2 تابع های رشته ای

در PL/SQL توابع زیادی بر روی رشته ها عمل می کنند. در جدول زیر لیست برخی از این توابع به همراه توضیحات آن را مشاهده می کنید. در این توابع منظور از علامت [] اختیاری بودن ورودی داخل آن است.

نام تابع	عملیات
ASCII(x);	مقدار ASCII برای کاراکتر x را برمی گرداند.
CHR(x);	کاراکتر متناظر مقدار ASCII که در x وارد می شود را برمی گرداند
CONCAT(x, y);	الحاق دو رشته x و yرا برمی گرداند.
INITCAP(x);	حرف اول هر کلمه در رشته x را بزرگ می کند.
INSTR(x, find_string [, start] [, occurrence]);	در رشته x رشته find_string را پیدا می کند و مکان آن را برمی گرداند.
INSTRB(x);	همانند INSTR است ولی مقدار برگشتی در قالب بایت است.
LENGTH(x);	تعداد کاراکترها در رشته x را برمی گرداند.
LENGTHB(x);	همانند LENTGH است ولی مقدار برگشتی در قالب بایت است.
LOWER(x);	تمام حروف رشته x را کوچک می کند.

LPAD(x, width [, pad_string]) ;	در سمت چپ رشته x به تعدادی که طول رشته x به WIDTH برسد رشته pad_string را درج می کند.
LTRIM(x [, trim_string]);	از سمت چپ رشته عمل حذف رشته ی trim_string را انجام می دهد.
NLS_INITCAP(x);	همانند INITCAP است. روش SORTING می تواند متفاوت باشد.
NLS_LOWER(x) ;	همانند LOWER است. روش SORTING می تواند متفاوت باشد.
NLS_UPPER(x);	همانند UPPER است. روش SORTING می تواند متفاوت باشد.
NLSSORT(x);	روش مرتب سازی کاراکترهای رشته را عوض می کند و قبل از استفاده از توابع NLS باید از این تابع استفاده شود.
NVL(x, value);	اگر x برابر با NULL باشد مقدار VALUE را برمی گرداند در غیر این صورت مقدار x برمی گردد.
NVL2(x, value1, value2);	اگر x برابر با NULL باشد مقدار VALUE1 را برمی گرداند در غیر این صورت مقدار VALUE2 برمی گردد.
REPLACE(x, search_string, replace_string);	رشته x را برای SEARCH_STRING جست و جو می کند تا با replace_string عوض کند.
RPAD(x, width [, pad_string]);	در سمت راست رشته x به تعدادی که طول رشته x به WIDTH برسد رشته pad_string را درج می کند.
RTRIM(x [, trim_string]);	از سمت راست رشته عمل حذف رشته ی trim_string را انجام می دهد.

SOUNDEX(x) ;	نمایش خروجی آوایی برای رشته x تعیین می کند.
SUBSTR(x, start [, length]);	رشته x را از محل start و به اندازه length جدا می کند. اگر length مشخص نشود مابقی رشته جدا می شود.
SUBSTRB(x);	همانند SUBSTR است ولی پارامترهای ورودی بر اساس بایت هستند.
TRIM(trim_char FROM x);	از سمت راست و چپ رشته x یک کاراکتر را که در trim_char مشخص می شود خذف می کند..
UPPER(x);	تمام حروف رشته x را بزرگ می کند.

مثال:

```
DECLARE
greetings varchar2(11) := 'hello world';
BEGIN
dbms_output.put_line(UPPER(greetings));
dbms_output.put_line(LOWER(greetings));
dbms_output.put_line(INITCAP(greetings));
/* retrieve the first character in the string */
dbms_output.put_line ( SUBSTR (greetings, 1, 1));
/* retrieve the last character in the string */
dbms_output.put_line ( SUBSTR (greetings, -1, 1));
/* retrieve five characters,
starting from the seventh position. */
dbms_output.put_line ( SUBSTR (greetings, 7, 5));
/* retrieve the remainder of the string,
starting from the second position. */
```

```
dbms_output.put_line ( SUBSTR (greetings, 2));
/* find the location of the first "e" */
dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

خروجی:

```
HELLO WORLD
hello world
Hello World
h
d
World
ello World
2
PL/SQL procedure successfully completed.
```

مثال:

```
DECLARE
greetings varchar2(30) := '......Hello World.....';
BEGIN
dbms_output.put_line(RTRIM(greetings,'.'));
dbms_output.put_line(LTRIM(greetings, '.'));
dbms_output.put_line(TRIM( '.' from greetings));
END;
/
```

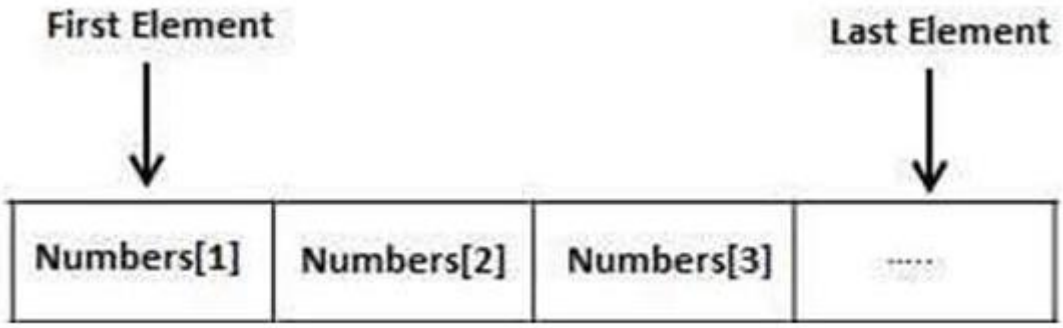
خروجی

```
......Hello World
Hello World.....
```

```
Hello World
PL/SQL procedure successfully completed.
```

6.3 آرایه

در زبان PL/SQL یک ساختمان داده یک بعدی به نام VARRAY وجود دارد که مجموعه ای از المنت های هم نوع با سایز ثابت را در کنار هم ذخیره کند. البته این ذخیره سازی در حافظه به صورت ترتیبی می باشد. یعنی به اولین المنت، آدرس اول از حافظه اختصاص می یابد و آخرین آدرس مربوط به المنت آخر است.



در ادامه سینتکس ساخت VARRAY را می بینید:

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

در این سینتکس n برابر با حداکثر تعداد المنت ها است و element_type نوع داده المنت های VARRAY است.

نکته: حداکثر سایز VARRAY یا همان n با دستور ALTER قابل تغییر می باشد.

مثال:

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/
Type created.
```

سینتکس ایجاد نوع داده:

```
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

مثال:

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

مثال:

```
DECLARE
     type namesarray IS VARRAY(5) OF VARCHAR2(10);
     type grades IS VARRAY(5) OF INTEGER;
     names namesarray;
     marks grades;
     total integer;
BEGIN
     names := namesarray('Kavita', 'Pritam', 'Ayan',
     'Rishav', 'Aziz');
     marks:= grades(98, 97, 78, 87, 92);
     total := names.count;
     dbms_output.put_line('Total '|| total || ' Students');
     FOR i in 1 .. total LOOP
          dbms_output.put_line('Student: ' || names(i) || '
          Marks: ' || marks(i));
     END LOOP;
END;
/
```

خروجی

```
Total 5 Students
Student: Kavita Marks: 98
Student: Pritam Marks: 97
Student: Ayan Marks: 78
Student: Rishav Marks: 87
Student: Aziz Marks: 92
PL/SQL procedure successfully completed.
```

نکته: VARRAY به صورت پیش فرض برابر با NULL است برای اینکه بتوان به المنت های آن رجوع کرد باید به آنها مقدار اختصاص یابد.

مثال:

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 511dfds | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+

DECLARE
     CURSOR c_customers is
     SELECT name FROM customers;
     type c_list is varray (6) of customers.name%type;
     name_list c_list := c_list();
     counter integer :=0;
BEGIN
     FOR n IN c_customers LOOP
          counter := counter + 1;
```

```
          name_list.extend;
          name_list(counter) := n.name;
          dbms_output.put_line('Customer('||counter
          ||'):'||name_list(counter));
     END LOOP;
END;
/
```

خروجی

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
PL/SQL procedure successfully completed.
```

7 پروسیجر در PL/SQL

در این فصل پروسیجر(PROCEDURE) و روش استفاده از آن را توضیح می دهیم ولی در ابتدا لازم است با مفهوم **زیربرنامه یا SUBPROGRAM** آشنا شویم. در زبان PL/SQL، زیربرنامه یک واحد از برنامه است که کار خاصی را انجام می دهد. برنامه های اصلی از ترکیب این زیربرنامه ها تشکیل می شوند و به این ترتیب مفهوم طراحی MODULAR شکل می گیرد. هر زیربرنامه می تواند بوسیله یک برنامه یا توسط یک زیربرنامه دیگر اجرا شود.

زیربرنامه های PL/SQL در واقع همان بلاک های PL/SQL هستند که برای آنها یک نام در نظر گرفته می شود و می توانند با تعدادی پارامتر فراخوانی شوند.

7.1 انواع زیربرنامه های PL/SQL :

- تابع(function) که فقط یک مقدار را بر می گرداند و معمولا از آنها برای محاسبات ریاضی و بازگشت نتیجه استفاده می شود.

- پروسیجر(procedure) که به طور مستقیم مقداری برنمی گرداند و از آنها برای اجرای عملیات خاص استفاده می شود.

7.2 کجا می توان یک زیربرنامه ساخت؟

- در سطح SCHEMA

به این نوع از زیربرنامه ها STANDALONE می گویند و در داخل دیتابیس اوراکل ذخیره می شوند. زیربرنامه های STANDALONE با دستور CREATE PROCEDURE یا CREATE FUNCTION ساخته می شوند و با دستور DROP می توان آنها را حذف نمود.

- درون یک پکیج

می توان در داخل پکیج یک زیربرنامه را ساخت و در دیتابیس ذخیره کرد ولی زمانی که آن پکیج حذف گردد این زیربرنامه نیز حذف می شود.

- در داخل یک بلاک PL/SQL

7.3 اجزای یک زیربرنامه در PL/SQL:

در فصل اول اجزای بلاک های ANONYMOUS را توضیح دادیم. همانند این بلاک ها، زیریرنامه ها هم شامل سه قسمت هستند:

1.قسمت declarative

استفاده از قسمت DECLARE اختیاری است البته برای زیربرنامه ها این قسمت با کلمه کلیدی DECLARE شروع **نمی شود**. در این قسمت متغیرها، CONSTANTها، نوع داده ها ، EXCEPTION و CRUSOR تعریف می شود. مواردی که در DECLARE تعریف می شوند فقط برای آن زیربرنامه به صورت محلی قابل دسترس است و با پایان اجرای زیربرنامه از بین می روند.

2.قسمت اجرایی

این قسمت اجباری است و شامل دستورات اجرایی آن زیربرنامه است.

3.قسمت EXCEPTION HANDLING

این قسمت اختیاری است و شامل دستوراتی می شود که در زمان رخداد خطای زیربرنامه، باید اجرا گردند.

7.4 ساخت پروسیجر

در ادامه سینتکس ساخت پروسیجر را می بینید:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
     < procedure_body >
END procedure_name;
```

در این سینتکس استفاده از پارامترها اختیاری است. همچنین معمولا زمانی که یک پروسیجر از نوع STANDALONE در داخل دیتابیس تعریف می شود بجای IS از کلمه کلیدی AS استفاده می گردد.

مثال: نمایش رشته HELLO WORLD توسط یک پروسیجر STANDALONE.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
     dbms_output.put_line('Hello World!');
END;

/
```

7.5 روش اجرای پروسیجرهای STANDALONE

به دو روش می توان پروسیجرهای STANDALONE را اجرا کرد.

1.با استفاده از کلمه کلیدی EXECUTE (یا استفاده از عبارت EXEC)

2.فراخوانی نام آن پروسیجر در داخل یک بلاک PL/SQL

مثال: اجرای پروسیجر GREETING به دو روش:

روش 1:

```
EXECUTE greetings;

------------------------------
```

روش 2:

```
BEGIN
     greetings;
END;
/
```

خروجی اجرای پروسیجر در این دو روش:

```
Hello World
PL/SQL procedure successfully completed.
```

7.6 پاک کردن پروسیجر STANDALONE:

برای حذف کردن یک پروسیجر باید از دستور DROP استفاده نمود.

```
DROP PROCEDURE greetings;
```

7.7 انواع پارامترهای زیربرنامه در PL/SQL

1.پارامتر از نوع IN

با استفاده از پارامتر IN می توان یک مقدار، عبارت، LITERAL ، CONSTANT یا متغیری که مقدار گرفته را به داخل زیربرنامه ارسال نمود. این پارامتر به صورت فقط خواندنی است و نمی توان آن را تغییر داد.

2.پارامتر از نوع OUT

این پارامتر یک مقدار را به برنامه فراخواننده بازمی گرداند. پارامتر OUT مانند یک متغیر عمل می کند که می توان مقدار آن را تغییر داد و دوباره به این مقدار جدید اشاره کرد. بنابراین پارامتر اصلی باید از نوع متغیر باشد که مقدار نهایی آن بازگردانده می شود.

3.پارامترهای از نوع IN OUT : این نوع پارامترها یک مقدار اولیه به داخل زیربرنامه ارسال می کنند ولی مقدار بروز شده را به فراخواننده بر می گردانند. همچنین پارامتر اصلی IN OUT باید از نوع متغیر باشد.

مثال 1: با پارامتر IN مقدارهای ورودی را دریافت کنید و مقدار حداقل آن را در پارامتر OUT قرار دهید.

```
DECLARE
     a number;
     b number;
     c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
     IF x < y THEN
          z:= x;
     ELSE
          z:= y;
     END IF;
END;
BEGIN
     a:= 23;
     b:= 45;
     findMin(a, b, c);
     dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

خروجی:

```
Minimum of (23, 45) : 23
PL/SQL procedure successfully completed.
```

نکته: در مثال بالا a، b و c **پارامترهای اصلی** و x، y و z **پارامترهای رسمی** هستند.

مثال 2: محاسبه و بازگرداندن توان دوم ورودی:

```
DECLARE
     a number;
     PROCEDURE squareNum(x IN OUT number) IS
     BEGIN
          x := x * x;
     END;
BEGIN
     a:= 23;
     squareNum(a);
     dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

خروجی

```
Square of (23): 529
PL/SQL procedure successfully completed.
```

7.8 روش های ارسال پارامتر

پارامترهای اصلی را می توان به سه روش به زیربرنامه ها ارسال نمود.

1.POSIIONAL NOTATION:

همان حالت عادی برای ارسال پارامترهای اصلی به پارامترهای رسمی است.

مثال:

```
findMin(a, b, c, d);
```

2.NAMED NOTATION:

به روش زیر و با علامت => می توان پارامتر اصلی را به پارامتر رسمی پاس داد.

```
findMin(x=>a, y=>b, z=>c, m=>d);
```

3. می توان ترکیبی از روش یک و دو استفاده کرد. **البته ابتدا باید از روش POSITIONAL استفاده نمود.**

مثال 1: این روش مجاز است:

```
findMin(a, b, c, m=>d);
```

مثال 2: این روش مجاز نیست:

```
findMin(x=>a, b, c, d);
```

8 تابع در PL/SQL

در این قسمت تابع (FUNCTION) در PL/SQL را توضیح می دهیم. تابع از هر لحاظ مانند پروسیجر است با این تفاوت که برخلاف پروسیجر، یک مقدار خاص برگردانده می شود. بنابراین تمام مطالبی که در **فصل هفتم آموزش PL/SQL** توضیح داده شد در مورد تابع نیز صادق است.

8.1 ایجاد تابع

با استفاده از دستور CREATE FUNCTION می توان یک تابع از نوع STANDALONE ایجاد کرد. در ادامه سینتکس ساخت تابع را می بینید.

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
    < function_body >
END [function_name];
```

این سینتکس همانند دستور ساخت پروسیجر است با این تفاوت که توسط دستور اجباری RETURN یک مقدار با نوع داده خاص به برنامه فراخواننده بازگردانی می شود.

مثال: تعداد کل مشتریان را از جدول COSTUMERS بشمارید و در متغیر TOTAL قرار دهید تا توسط تابع totalCustomers برگردانده شود.

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
total number(2) := 0;
BEGIN
     SELECT count(*) into total
     FROM customers;
     RETURN total;
END;
/
```

```
Function created.
```

مثال: تابع total_costumer() را از یک بلاک ANONYMOUS فراخوانی کنید و مقدار برگشتی را در متغیر C ذخیره کنید.

```
DECLARE
     c number(2);
BEGIN
     c := totalCustomers();
     dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

خروجی:

```
Total no. of Customers: 6
PL/SQL procedure successfully completed.
```

مثال: بین دومقدار ورودی، مقدار حداکثر را محاسبه کنید و نمایش دهید.

```
DECLARE
     a number;
     b number;
     c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
     z number;
     BEGIN
          IF x > y THEN
```

```
            z:= x;
        ELSE
            Z:= y;
        END IF;
        RETURN z;
    END;

BEGIN
    a:= 23;
    b:= 45;
    c := findMax(a, b);
    dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

خروجی:

```
Maximum of (23,45): 45
PL/SQL procedure successfully completed.
```

8.2 توابع از نوع RECURSIVE یا بازگشتی

همانطور که می دانید یک برنامه یا زیربرنامه می تواند زیربرنامه های دیگر را فراخوانی کند ولی زمانی که یک زیربرنامه خودش را فراخوانی می کند به این عمل RECURSIVE می گویند.

مثال: فاکتوریل عدد ورودی n را محاسبه کنید و برگردانید.

```
n! = n*(n-1)!
= n*(n-1)*(n-2)!
...
= n*(n-1)*(n-2)*(n-3)... 1

DECLARE
num number;
factorial number;
FUNCTION fact(x number)
RETURN number
IS
f number;

BEGIN
IF x=0 THEN
f := 1;
ELSE
f := x * fact(x-1);
END IF;
RETURN f;
END;
BEGIN
num:= 6;
factorial := fact(num);
dbms_output.put_line(' Factorial '|| num || ' is ' ||
factorial);
```

```
END;
/
```

خروجی:

```
Factorial 6 is 720
PL/SQL procedure successfully completed.
```

9 CURSOR در PL/SQL

در این فصل CRUSOR و روش استفاده از آن را توضیح می دهیم. دیتابیس اوراکل برای هر دستور SQL یک فضای حافظه به نام CONTEXT AREA ایجاد می کند که اطلاعات پردازشی دستور در این محل ذخیره می شود. CURSOR یک **اشاره گر** به CONTEXT AREA است و PL/SQL با استفاده از CURSOR این فضا را کنترل می کند و اطلاعات آن را نمایش می دهد. برای مثال با استفاده از CURSOR می توان تعداد کل سطرهای دستکاری شده توسط یک DML را مشاهده نمود یا محتویات داده در هر سطر از یک QUERY را بدست آورد.

CURSORها می توانند ضمنی یا صریح باشند که در ادامه توضیح داده می شوند.

9.1 CURSORهای از نوع IMPLICIT یا ضمنی

هر زمان که یک دستور DML یا یک دستور SELECT INTO توسط اوراکل اجرا می شود به صورت اتوماتیک و به روش ضمنی یک CURSOR برای آن دستور ایجاد می شود. برای دستورات INSERT محتویات CURSOR شامل اطلاعاتی است که باید درج شود و برای دستورات UPDATE و DELETE سطرهایی که مورد تاثیر قرار می گیرند توسط آن CURSOR شناسایی می شوند.

برنامه نویسان نمی توانند این نوع از CURSORها و اطلاعات آنها را کنترل کنند ولی می توانند اطلاعات و وضعیت CURSOR را از طریق ویژگی ها جدول زیر مشاهده کنند. با استفاده از این ویژگی ها که با SQL آغاز می شود می توان به اطلاعات آخرین CURSOR از نوع ضمنی دست یافت.

ویژگی	توضیحات
SQL%FOUND	**اگر یک دستور DML یا SELECT INTO یک یا بیشتر ار یک سطر را تحت تاثیر قرار دهد مقدار TRUE را برمی گرداند در غیر این صورت مقدار FALSE بازمی گردد**
SQL%NOTFOUND	**از لحاظ منطق برعکس %FOUND عمل می کند.**
SQL%ISOPEN	**برای CURSORهای از نوع ضمنی همیشه برابر با FALSE است زیرا بلافاصله بعد از اجرای دستور، CRUSOR از نوع ضمنی توسط اوراکل CLOSE می شود.**
SQL%ROWCOUNT	**تعداد سطرهایی که توسط دستورات DML دستکاری شده اند یا توسط دستور SELECT INTO برگردانده شده اند را ذخیره می کند.**

مثال: جدول مشتریان به این شکل می باشد:

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

حال برنامه زیر را اجرا می کنیم:

```
DECLARE
        total_rows number(2);
BEGIN
        UPDATE customers
        SET salary = salary + 500;
        IF sql%notfound THEN
                dbms_output.put_line('no customers selected');
        ELSIF sql%found THEN
                total_rows := sql%rowcount;

                dbms_output.put_line( total_rows || ' customers selected ');
        END IF;
END;
/
```

در این برنامه به حقوق تمام مشتریان 500 واحد اضافه شده است. بنابراین CURSOR که به صورت ضمنی تعریف شده است دارای مقدار صحیح برای ویژگی %FOUND است و ویژگی %ROWCOUNT آن نیز برابر با تعداد سطرهای UPDATE شده خواهد بود.

خروجی:

```
6 customers selected
PL/SQL procedure successfully completed.
```

خروجی جدول:

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

نکته: با توجه به ماهیت CURSOR از نوع ضمنی و زمانی که صفر یا چند سطر توسط دستور برگردانده می شود ممکن است به خطای EXCEPTION از نوع NO_DATA_FOUND یا TOO_MANY_ROWS برخورد کنیم.

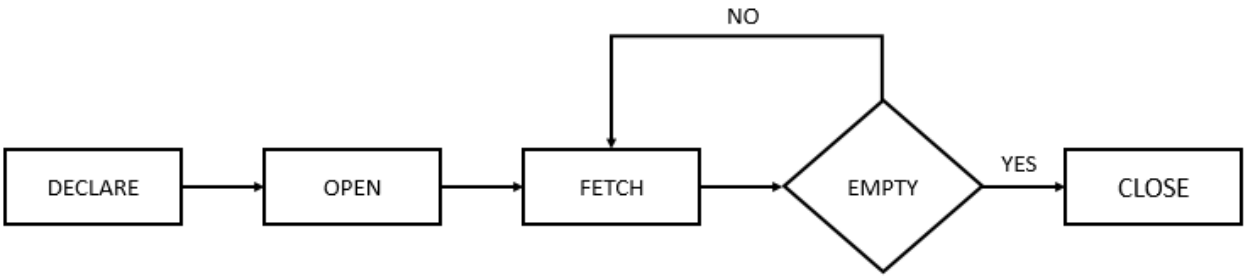
9.2 CURSOR از نوع صریح یا EXPLICIT:

این نوع از CURSORها توسط برنامه نویسان و به منظور کنترل اطلاعات CONTEXT AREA ایجاد می شود. CURSORهای از نوع صریح در قسمت DECLARE بلاک برنامه تعریف می شوند. هر CURSOR از نوع صریح برای یک دستور SELECT که بیشتر از یک سطر را برمی گرداند ایجاد می شود.

در ادامه سینتکس ساخت CURSOR از نوع صریح را می بینید:

```
CURSOR cursor_name IS select_statement;
```

زمانی که از CURSOR به روش صریح استفاده می شود دیتابیس اوراکل یک سیکل اجرایی برای آن CURSOR تعریف می کند. در ادامه شکل مربوط به این سیکل اجرایی را می بینید و هر مرحله از این سیکل را توضیح می دهیم.



مرحله DECLARE:

ابتدا باید در قسمت DECLARE یک بلاک یا پکیج به شکل زیر CURSOR را تعریف کنیم:

```
CURSOR c_customers IS
SELECT id, name, address FROM customers;
```

مرحله OPEN:

با استفاده از دستور زیر یک CURSOR از نوع صریح OPEN می شود و از این مرحله به بعد آماده دستیابی خواهد بود.

```
OPEN c_customers;
```

بعد از OPEN شدن یک CURSOR اوراکل دستور SELECT مربوط به آن را پارس کرده و سپس اجرا می کند و CURSOR را روی اولین سطری که آن دستور برمی گرداند تنظیم می کند. در این مرحله CONTEXT AREA تشکیل می شود.

مرحله FETCH

در این مرحله به یک سطر از دستور توسط CURSOR اشاره می شود و مقدارهای این سطر در متغیرها ذخیره می شوند.

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

هر FETCH یک سطر را دستیابی می کند و با FETCH بعدی سطر بعدی دستیابی می شود.

مرحله CLOSE

با CLOSE کردن یک CURSOR فضای CONTEXT AREA آزاد می شود.

```
CLOSE c_customers;
```

نکته: اگر اجرای یک بلاک از نوع ANONYMOUS، تابع یا پروسیجرخاتمه یاید به صورت اتوماتیک CURSORهای از نوع صریح CLOSE می شوند.

نکته: اگر یک CURSOR که هنوز OPEN نشده است را CLOSE کنیم خطای EXCEPTION از نوع INVALID_CURSOR دریافت می کنیم.

نکته: ویژگی هایی که می توان بعد از نام یک CURSOR از نوع صریح استفاده کرد را در جدول زیر می بینید:

ویژگی	توضیحات
%FOUND	**1.قبل از مرحله FETCH برابر با NULL است.** **2.در زمان FETCH شدن یک سطر TRUE است.** **3.اگر هیچ سطر دیگری برای FETCH کردن وجود نداشته باشد برابر با FALSE است.** **4. اگر CURSOR هنوز OPEN نشده باشد برابر با INVALID_CURSOR است.**
%NOTFOUND	**1.قبل از مرحله FETCH برابر با NULL است.** **2.در زمان FETCH شدن یک سطر، FALSE است.** **3.اگر هیچ سطر دیگری برای FETCH کردن وجود نداشته باشد برابر با TRUE است.** **4. اگر CURSOR هنوز OPEN نشده باشد برابر با INVALID_CURSOR است.**
%ISOPEN	**اگر CURSOR از نوع صریح OPEN باشد مقدار TRUE برمی گرداند در غیر این صورت مقدار آن FALSE است.**
%ROWCOUNT	**تعداد سطرهایی که FETCH شده است را برمی گرداند. اگر هنوز OPEN نشده باشد INVALID_CURSOR را بر می گرداند**

مثال: نام و شماره تمام مشتریان را نمایش دهید.

```
DECLARE
      c_id customers.id%type;
      c_name customerS.Name%type;
      c_addr customers.address%type;
      CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
      OPEN c_customers;
      LOOP
            FETCH c_customers into c_id, c_name, c_addr;
            EXIT WHEN c_customers%notfound;
            dbms_output.put_line(c_id || ' ' || c_name || ' ' ||
            c_addr);
      END LOOP;
      CLOSE c_customers;
END;
/
```

در این برنامه تا زمانی که ویژگی %NOTFOUND برای CURSOR برقرار نشده است، نام و شماره مشتریان چاپ می شود.

خروجی:

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP
PL/SQL procedure successfully completed.
```

نکته: می توان به CURSORهای از نوع صریح، پارامتر ارسال نمود. بنابراین می توان هر زمان آن CURSOR را با آرگومنت های متفاوت OPEN کنیم.

مثال: یک مرتبه CURSOR را با حداقل و حداکثر قیمت بین 50 و 100 و بار دیگر با 800 و 1000 درنظر می گیریم..

```
DECLARE
        r_product products%rowtype;
        CURSOR c_product (low_price NUMBER, high_price NUMBER)
         IS
         SELECT *
         FROM products
         WHERE list_price BETWEEN low_price AND high_price;
BEGIN
        -- show mass products
        dbms_output.put_line('Mass products: ');
        OPEN c_product(50,100);
        LOOP
            FETCH c_product INTO r_product;
            EXIT WHEN c_product%notfound;
            dbms_output.put_line(r_product.product_name || ': ' ||r_product.list_price);
         END LOOP;
         CLOSE c_product;

        -- show luxury products
        dbms_output.put_line('Luxury products: ');
        OPEN c_product(800,1000);
        LOOP
                FETCH c_product INTO r_product;
                EXIT WHEN c_product%notfound;
                dbms_output.put_line(r_product.product_name || ': ' ||r_product.list_price);
        END LOOP;
        CLOSE c_product;

END;
/
```

10 رکورد در PL/SQL

در این فصل رکورد (RECORD) در PL/SQL و روش استفاده از آن را توضیح می دهیم. رکورد یک ساختمان داده است که آیتم های متفاوت را کنار هم نگه می دارد. یعنی بر خلاف آرایه، این آیتم ها نوع داده متفاوت دارند. به همین جهت، هر سطر از جدول را یک رکورد می نامند زیرا معمولا سطرها از ستون های با نوع داده متفاوت تشکیل شده اند.

برای مثال می خواهیم اطلاعات کتاب های یک کتابخانه را بررسی کنیم. برای این کتاب ها مشخصات مختلف از جمله عنوان، نام نویسنده، موضوع کتاب و شماره کتاب وجود دارد. بنابراین برای آن ها یک رکورد با فیلدهای مورد نیاز تعریف می کنیم و اطلاعات مورد نظر را به شیوه بهتر دسته بندی کرده و نمایش می دهیم.

در PL/SQL می توان رکورد را به سه روش زیر تعریف کرد که در ادامه توضیح می دهیم.

10.1 روش Table-Based

در روش Table-Based از ویژگی %ROWTYPE استفاده می شود و رکورد معادل با ستون ها و نوع داده آنها در یک جدول ساخته می شود. برای دسترسی به محتویات رکورد نیز از علامت نقطه (.) استفاده می شود.

مثال:

```
DECLARE
    customer_rec customers%rowtype;
BEGIN
    SELECT * into customer_rec
    FROM customers
    WHERE id = 5;
    dbms_output.put_line('Customer ID: ' || customer_rec.id);
    dbms_output.put_line('Customer Name: ' || customer_rec.name);
    dbms_output.put_line('Customer Address: ' || customer_rec.address);
```

```
    dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/
```

خروجی:

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000
PL/SQL procedure successfully completed.
```

10.2 روش Cursor-Based

همانند روش اول در تعریف یک رکورد از ویژگی %ROWTYPE استفاده می شود ولی ستون ها و نوع داده آنها معادل با یک CURSOR تعریف می شود. با استفاده از CURSOR می توان به رکوردهای مختلف دسترسی داشته باشیم.

مثال:

```
DECLARE
    CURSOR customer_cur is
    SELECT id, name, address
    FROM customers;
    customer_rec customer_cur%rowtype;
BEGIN
    OPEN customer_cur;
    LOOP
        FETCH customer_cur into customer_rec;
        EXIT WHEN customer_cur%notfound;
```

```
            DBMS_OUTPUT.put_line(customer_rec.id || ' ' ||
            customer_rec.name);
        END LOOP;
END;
/
```

خروجی:

```
1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal
PL/SQL procedure successfully completed.
```

10.3 .روش User-Defined

در این روش کاربر می تواند یک رکورد با فیلدها و نوع داده دلخواه تعریف کند. در ادامه سینتکس تعریف رکورد به این روش را می بینید:

```
 TYPE
type_name IS RECORD
( field_name1 datatype1 [NOT NULL] [:= DEFAULT EXPRESSION],
field_name2 datatype2 [NOT NULL] [:= DEFAULT EXPRESSION],
...
field_nameN datatypeN [NOT NULL] [:= DEFAULT EXPRESSION);

record-name type_name;
```

مثال 1: تعریف رکورد برای دو کتاب

```
DECLARE
TYPE books IS RECORD
(title varchar(50),
author varchar(50),
subject varchar(100),
book_id number);
book1 books;
book2 books;
```

مثال 2: ثبت و نمایش اطلاعات دو کتاب در رکورد

```
DECLARE
type books is record
(title varchar(50),
author varchar(50),
subject varchar(100),
book_id number);
book1 books;
book2 books;
BEGIN
-- Book 1 specification
book1.title := 'C Programming';
book1.author := 'Nuha Ali ';
book1.subject := 'C Programming Tutorial';
book1.book_id := 6495407;
-- Book 2 specification
book2.title := 'Telecom Billing';
book2.author := 'Zara Ali';
book2.subject := 'Telecom Billing Tutorial';
book2.book_id := 6495700;
-- Print book 1 record
```

```
dbms_output.put_line('Book 1 title : '|| book1.title);
dbms_output.put_line('Book 1 author : '|| book1.author);
dbms_output.put_line('Book 1 subject : '|| book1.subject);
dbms_output.put_line('Book 1 book_id : ' || book1.book_id);
-- Print book 2 record
dbms_output.put_line('Book 2 title : '|| book2.title);
dbms_output.put_line('Book 2 author : '|| book2.author);
dbms_output.put_line('Book 2 subject : '|| book2.subject);
dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

خروجی:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
PL/SQL procedure successfully completed.
```

10.4 ارسال رکورد به عنوان پارامتر

می توان یک رکورد را همانند متغیر به زیربرنامه ها ارسال نمود و در داخل آن زیربرنامه به فیلدهای آن رکورد دسترسی یافت.

مثال:

```
DECLARE  type books is record
(title varchar(50),
author varchar(50),
subject varchar(100),
book_id number);
book1 books;
book2 books;
PROCEDURE printbook (book books) IS
BEGIN
dbms_output.put_line ('Book title : ' || book.title);
dbms_output.put_line('Book author : ' || book.author);
dbms_output.put_line( 'Book subject : ' || book.subject);
dbms_output.put_line( 'Book book_id : ' || book.book_id);
END;
BEGIN
-- Book 1 specification
book1.title := 'C Programming';
book1.author := 'Nuha Ali ';
book1.subject := 'C Programming Tutorial';
book1.book_id := 6495407;
 -- Book 2 specification
book2.title := 'Telecom Billing';
book2.author := 'Zara Ali';
book2.subject := 'Telecom Billing Tutorial';
book2.book_id := 6495700;
-- Use procedure to print book info
printbook(book1);
printbook(book2);
END;
/
```

خروجی:

Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
PL/SQL procedure successfully completed.

11 EXCEPTION و EXCEPTION-HANDLER در PL/SQL

در این فصل EXCEPTION و روش استفاده از EXCEPTION-HANDLER را توضیح می دهیم. منظور از EXCEPTION رخداد خطا در زمان اجرای برنامه است. برای مثال به دنبال اطلاعاتی هستیم که در دیتابیس وجود ندارند یا یک ورودی اشتباه به زیربرنامه ارسال می کنیم. زبان PL/SQL با استفاده از بلاک EXCEPTION-HANDLER به برنامه نویسان کمک می کند تا نوع EXCEPTION برنامه را متوجه شوند و در زمان رخداد آنها دستورات مناسب را اجرا کنند.

در زبان PL/SQL دو دسته EXCEPTION داریم:

1. SYSTEM-DEFINED EXCEPTIONS: خطاهایی که رخداد آنها توسط دیتابیس قابل تشخیص است.

2. USER-DEFINED EXCEPTONS: خطاهایی که توسط برنامه نویسان تعریف می شوند.

در برنامه های PL/SQL خطاها یا همان EXCEPTIONها با استفاده از EXCEPTION-HANDLER مدیریت می شوند. در ادامه سینتکس تعریف بلاک EXCEPTION-HANDLER را در داخل یک برنامه مشاهده می کنید. در این سینتکس در زمان رخداد خطای exceptionN، دستورهای exception-handling-statements اجرا می شوند. همچنین قسمت WHEN OTHERS مربوط به خطاهایی است که نوع آنها در تعریف EXCEPTION-HANDLER مشخص نشده است.

```
DECLARE
<declarations section>
BEGIN
        <executable command(s)>
EXCEPTION
    <exception handling goes here >
    WHEN exception1 THEN
```

```
        exception1-handling-statements
        WHEN exception2 THEN
        exception2-handling-statements
        WHEN exception3 THEN
        exception3-handling-statements
        ........
        WHEN others THEN
        exception3-handling-statements
END;
```

مثال: اطلاعات مشتری شماره 8 وجود ندارد بنابراین در زمان اجرای برنامه خطای از نوع -SYSTEM DEFINED رخ می دهد و مقدار NO_DATA_FOUND صحیح می شود..

```
DECLARE
        c_id customers.id%type := 8;
        c_name customerS.name%type;
        c_addr customers.address%type;
BEGIN
        SELECT name, address INTO c_name, c_addr
        FROM customers
        WHERE id = c_id;
        DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
        DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
        WHEN no_data_found THEN
                dbms_output.put_line('No such customer!');
        WHEN others THEN
                dbms_output.put_line('Error!');
END;
/
```

خروجی:

No such customer!
PL/SQL procedure successfully completed.

11.1 خطاهای از نوع USER-DEFINED

بعضی از خطاها که در حین اجرای برنامه رخ می دهند، توسط دیتابیس قابل تشخیص هستند و به آنها SYSTEM-DEFINED EXCEPTION می گویند. مانند خطای NO_DATA_FOUND در مثال قبل. اما برنامه نویسان می توانند بر اساس نیاز برنامه یک خطای خاص را تعریف کنند و آن را با استفاده از بلاک HANDLER کنترل کنند. این دسته از خطاها ابتدا باید در قسمت DECLARE برنامه تعریف شوند و سپس رخداد آنها به صورت صریح و با کلمه کلیدی RAISE مشخص شود.

نکته: می توان بجای کلمه کلیدی RAISE از پروسیجر زیر استفاده نمود:

DBMS_STANDARD.RAISE_APPLICATION_ERROR

مثال: شماره مشتری به عنوان ورودی از کاربر پرسیده می شود و اگر این شماره کوچکتر از صفر وارود شود خطای ex_invalid_id رخ می دهد. همچنین در این مثال خطای سیستمی NO_DATA_FOUND نیز در بلاک HANDLER مشخص شده است.

```
DECLARE
        c_id customers.id%type := &cc_id;
        c_name customerS.name%type;
        c_addr customers.address%type;
        -- user defined exception
        ex_invalid_id EXCEPTION;
```

```
BEGIN
        IF c_id <= 0 THEN
                RAISE ex_invalid_id;
        ELSE
                SELECT name, address INTO c_name, c_addr
                FROM customers
                WHERE id = c_id;
                 DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
                DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
        END IF;
EXCEPTION
        WHEN ex_invalid_id THEN
                dbms_output.put_line('ID must be greater than zero!');
        WHEN no_data_found THEN
                dbms_output.put_line('No such customer!');
        WHEN others THEN
                dbms_output.put_line('Error!');
END;
/
```

خروجی(ورودی از طرف کاربر برابر با منفی شش است)

```
Enter value for cc_id: -6 (let's enter a value -6)
old 2: c_id customers.id%type := &cc_id;
new 2: c_id customers.id%type := -6;
ID must be greater than zero!
PL/SQL procedure successfully completed.
```

11.2 خطاهای از نوع SYSTEM-DEFINED

این دسته از خطاها توسط دیتابیس قابل تشخیص هستند و به صورت اتوماتیک RAISE آنها تشخیص داده می شود. در جدول زیر برخی از خطاهای سیستمی مهم را می بینید.

نوع EXCEPTION	خطای ORACLE	کد SQL	زمان رخداد
ACCESS_INTO_NULL	**06530**	**-6530**	**زمانی که به یک OBJECT قبل از اینکه مقداردهی اولیه شود مقدار اختصاص دهیم.**
CASE_NOT_FOUND	**06592**	**-6592**	اگر هیچ کدام از عبارات WHEN در دستور CASE انتخاب نشوند و در آن دستور ELSE نیز نداشته باشیم
DUP_VAL_ON_INDEX	**00001**	**-1**	اگر قصد داشته باشیم روی یک INDEX از نوع UNIQUE اطلاعات تکراری قرار دهیم.
INVALID_CURSOR	**01001**	**-1001**	عملیات اشتباه روی CURSOR برای مثال زمانی که یک CURCSOR که هنوز OPEN نشده را CLOSE کنیم.
INVALID_NUMBER	**01722**	**-1722**	تبدیل اطلاعات از رشته کاراکتری به نوع داده عددی امکان پذیر نیست.
LOGIN_DENIED	**01017**	**-1017**	USER/PASS برای اتصال به دیتابیس اشتباه است.
NO_DATA_FOUND	**01403**	**+100**	زمانی که یک دستور SELECT INTO هیچ سطری را برنمی گرداند.
NOT_LOGGED_ON	**01012**	**-1012**	بدلیل عدم اتصال به دیتابیس فراخوانی از دیتابیس امکان پذیر نیست

PROGRAM_ERROR	06501	-6501	PL/SQL یک مشکل داخلی دارد.
ROWTYPE_MISMATCH	06504	-6504	زمانی که یک CURSOR اطلاعاتی را برای یک متغیر FETCH می کند که نوع داده متفاوت دارد.
STORAGE_ERROR	06500	-6500	زمانی که دچار کمبود حافظه می شویم.
TOO_MANY_ROWS	01422	-1422	زمانی که یک دستور SELECT INTO بیشتر از یک سطر را برمی گرداند.
VALUE_ERROR	06502	-6502	در عملیات ریاضی یا تبدیل نوع دچار مشکل شویم یا سایز داده باعث خطا شود.
ZERO_DIVIDE	01476	1476	زمانی که در برنامه یک عدد تقسیم بر صفر می شود

نکته: توابع SQLCODE و SQLERRM کد SQL و کد مربوط به خطای اوراکل به همراه توضیحات آن را بر می گردانند.

مثال: کارمند با شماره پرسنلی 1000 وجود ندارد.

```
DECLARE
  name employees.last_name%TYPE;
  v_code NUMBER;
  v_errm VARCHAR2(64);
BEGIN
  SELECT last_name INTO name FROM employees WHERE employee_id = 1000;
  EXCEPTION
    WHEN OTHERS THEN
      v_code := SQLCODE;
      v_errm := SUBSTR(SQLERRM, 1 , 64);
```

```
        DBMS_OUTPUT.PUT_LINE('The error code is ' || v_code || '- ' || v_errm);
END;
/
```

خروجی:

```
The error code is 100- ORA-01403: no data found
```

12 TRIGGER در PL/SQL

در این فصل TRIGGER و روش استفاده از آن را توضیح می دهیم. TRIGGER یک بلاک برنامه ذخیره شده در دیتابیس اوراکل است که همزمان با رخدادهای خاص به صورت اتوماتیک اجرا (FIRE) می شود. رخدادهایی که سبب اجرای یک TRIGGER می شوند عبارتند از:

1.رخداد دستورات DML

2.رخداد دستورات DDL

3.رخدادهای خاص در دیتابیس مانند LOGON، LOGOFF، STARTUP و SHUTDOWN

12.1 کجا می توان TRIGGER را تعریف نمود؟

- جدول
- VIEW
- در سطح SCHEMA
- در سطح دیتابیس

12.2 چرا از TRIGGER استفاده می شود؟

- جلوگیری از تراکنش های اشتباه
- ایجاد امنیت
- انجام REPLICATION همزمان در جدول ها
- اطلاعات مربوط به دسترسی یا دستکاری ذخیره شوند.
- تولید برخی مقدارهای خاص به صورت اتوماتیک

12.3 ساخت TRIGGER برای دستورات DML

در ادامه سینتکس ساخت TRIGGER برای دستورات DML را می بینید.

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
        Declaration-statements
BEGIN
        Executable-statements
EXCEPTION
        Exception-handling-statements
END;
```

- با استفاده از BEFOREیا AFTER می توان مشخص نمود که زمان اجرا شدن TRIGGER قبل یا بعد از رخداد باشد. عبارت INSTEAD OF برای ساخت TRIGGER بر روی VIEW استفاده می شود.
- در قسمت **{INSERT [OR] | UPDATE [OR] | DELETE}** عملیات DML مورد نظر را مشخص می کنیم و عبارت `[OF col_name]` نام ستونی که دستکاری می شود را مشخص می کند. می توان یک یا چند DML را مشخص نمود.
- قسمت `ON table_name` نام جدولی است که TRIGGER برای آن اجرا می شود
- با استفاده از **[REFERENCING OLD AS o NEW AS n]** یعنی n و o می توان به مقدارهای قبل و بعد از دستور DML اشاره نمود. اگر از این عبارت استفاده نشود از OLD و NEW می توان استفاده کرد. همچنین توجه شود که این مقدارها فقط برای TRIGGERهای از نوع ROW-LEVEL قابل استفاده هستند.

- اگر از عبارت **[FOR EACH ROW]** استفاده گردد TRIGGER برای هر سطر که دستکاری شده است اجرا می شود (ROW-LEVEL TRIGGER). در غیر این صورت TRIGGER فقط یکبار برای تمام دستور DML اجرا می شود(TABLE-LEVEL TRIGGER).
- اگر TRIGGER به صورت ROW-LEVEL تعریف شود دستور شرطی WHEN برای هر سطر بررسی می شود در غیر این صورت نمی توان از WHEN استفاده تمود.

نکته: نمی توان برای OBJECTهایی که کاربر SYS و SYSTEM مالک آنها هستند TRIGGER تعریف کرد.

مثال 1: یک TRIGGER به صورت ROW-LEVEL برای هرگونه عملیات DML در جدول costumers بسازید.

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
	sal_diff number;
BEGIN
	sal_diff := :NEW.salary - :OLD.salary;
	dbms_output.put_line('Old salary: ' || :OLD.salary);
	dbms_output.put_line('New salary: ' || :NEW.salary);
	dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

با اجرای دستور DML زیر این TRIGGER را FIRE می کنیم.

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

خروجی:

Old salary:
New salary: 7500
Salary difference:

مقدار OLD برای دستور قبلی برابر با NULL بود. در ادامه، یک دستور UPDATE اجرا می کنیم.

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

خروجی

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

نکته: توجه شود که از علامت **:** قبل از دستیابی به مقدارهای NEW و OLD استفاده می شود.

مثال 2: هر عمل INSERT در جدول ORDERS توسط چه شخصی و در چه تاریخی انجام می شود؟ این اطلاعات را در همان سطر از جدول و در ستونهای CREATED_BY و CREATE_DATE ذخیره کنید.

```
CREATE OR REPLACE TRIGGER orders_before_insert
BEFORE INSERT
  ON orders
  FOR EACH ROW
DECLARE
```

```
  v_username varchar2(10);
BEGIN
  -- Find username of person performing INSERT into table
  SELECT user INTO v_username   FROM dual;
  -- Update create_date field to current system date
  :new.create_date := sysdate;
  -- Update created_by field to the username of the person performing the
INSERT
  :new.created_by := v_username;
END;
/
```

نکته: در داخل برنامه TRIGGER از نوع BEFORE می توان مقدارهای NEW را عوض کرد ولی مقدارهای OLD قابل تغییر نیستند. برای مثال می توان از دستور زیر استفاده نمود:

```
:new.salary:=200;
```

نکته: در داخل TRIGGER های از نوع AFTER هیچ کدام از مقدارهای NEW و OLD قابل تغییر نیستند.

نکته: اگر می خواهیم در داخل برنامه TRIGGER که برای جدول A تعریف شده است، جدول A را پرس و جو کنیم می بایست از TRIGGER از نوع AFTER استفاده شود در غیر این صورت نمی توان از آن جدول QUERY گرفت.

12.4 ساخت TRIGGER برای دستورات DDL

برای دستورات DDL نیز می توان TRIGGER ساخت. در ادامه سینتکس ساخت TRIGGER برای دستورات DDL را می بینید که در سطح دیتابیس یا SCHEMA اعمال می شوند.

```
CREATE [OR REPLACE] TRIGGER trigger name
{BEFORE | AFTER } { DDL event} ON {DATABASE | SCHEMA}
[WHEN (...)]
DECLARE
 Variable declarations
BEGIN
...some code...
END;
```

مثال: در سطح SCHEMA اطلاعات هر دستور CREATE را ثبت کنید

```
CREATE OR REPLACE TRIGGER bcs_trigger
BEFORE CREATE
ON SCHEMA
DECLARE
 oper ddl_log.operation%TYPE;
BEGIN
 INSERT INTO ddl_log
 SELECT ora_sysevent, ora_dict_obj_owner,
 ora_dict_obj_name, NULL, USER, SYSDATE
 FROM DUAL;
END bcs_trigger;
/
```

12.5 ساخت TRIGGER برای رخدادهای خاص

در ادامه سینتکس ساخت TRIGGER برای رخدادهای خاص در سطح دیتابیس یا SCHEMA را ملاحظه می کنید:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} {database_event} ON {DATABASE | SCHEMA}
BEGIN
   PL/SQL Code
END;
/
```

مثال: عمل LOGON کاربران به دیتابیس را ثبت کنید.

```
CREATE OR REPLACE TRIGGER   all_lgon_audit
AFTER LOGON ON DATABASE
BEGIN
  INSERT INTO tbl_evnt_audit VALUES(
    ora_sysevent,
    sysdate,
    TO_CHAR(sysdate, 'hh24:mi:ss'),
    USER,
    NULL
  );
  COMMIT;
END;
/
```

12.6 حذف TRIGGER

برای حذف کردن یک TRIGGER از دستور DROP استفاده می شود:

```
DROP TRIGGER trigger_name;
```

12.7 فعال یا غیر فعال کردن یک TRIGGER

برای غیرفعال کردن TRIGGER از دستور زیر استفاده می شود:

```
ALTER TRIGGER orders_before_insert DISABLE;
```

اگر بخواهیم تمامی TRIGGER هایی که روی جدول تعریف شده است را غیر فعال کنیم از دستور زیر استفاده می کنیم:

```
ALTER TABLE table_name DISABLE ALL TRIGGERS;
```

دستور فعال کردن TRIGGER به این شکل است:

```
ALTER TABLE table_name DISABLE ALL TRIGGERS;
```

تمام TRIGGERهای یک جدول به این روش فعال می شوند:

```
ALTER TABLE table_name ENABLE ALL TRIGGERS;
```

13 پکیج در PL/SQL

در این فصل پکیج (PACKAGE) را توضیح می دهیم. پکیج یکی از OBJECT های دیتابیس اوراکل است که متغیرها، زیربرنامه ها و ... که از لحاظ منطقی به هم مرتبط هستند را در یک گروه قرار می دهد.

برای هر پکیج دو قسمت تعریف می شود:

1.قسمت مشخصات پکیج یا PACKAGE SPECIFICATION

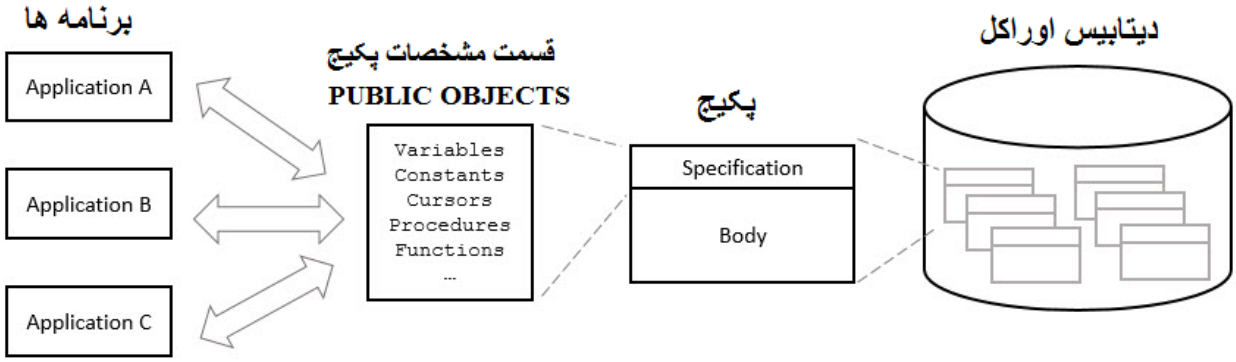
2.قسمت بدنه پکیج یا PACKAGE BODY(DEFINITION)

در ادامه این دو قسمت را توضیح می دهیم.

13.1 قسمت مشخصات پکیج

قسمت مشخصات یا SPEC یک واسط (INTERFACE) به محیط بیرون از پکیج است و استفاده از آن برای هر پکیج اجباری است. در این قسمت OBJECTهایی مانند متغیر، نوع داده ، CONSTANT، EXCEPTION، CURSOR و زیربرنامه تعریف می شوند که می توان از بیرون پکیج آنها را فراخوانی کرد. در واقع قسمت SPEC، شامل اطلاعات اصلی پکیج به جز کد مربوط به زیربرنامه ها و CURSORها است.

تمام OBJECTهایی که در قسمت مشخصات پکیج قرار دارند و از محیط بیرون قابل دسترس هستند PUBLIC نامیده می شوند ولی هر OBJECT که در این قسمت نباشد و در قسمت بدنه پکیج تعریف شده باشد یک OBJECT از نوع PRIVATE نامیده می شود و فقط در حوزه همان پکیج قابل اجرا خواهد بود.



مثال: قسمت مشخصات پکیج برای پکیج `cust_sal` را ایجاد کنید که شامل یک زیربرنامه به نام `find_sal` است.

```
CREATE PACKAGE cust_sal AS
PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/

Package created.
```

13.2 قسمت بدنه پکیج

اگر در قسمت مشخصات یا SPEC یک پکیج از زیربرنامه یا CURSOR استفاده شده باشد می بایست از قسمت بدنه استفاده نمود تا کد برنامه مربوط به آنها در پکیج تعریف شود. در این قسمت می توان OBJECT هایی مانند متغیرها، زیربرنامه ها و ... تعریف نمود که در قسمت مشخصات پکیج نوشته نشده باشند ولی این OBJECTها از محیط بیرون از بدنه پکیج قابل دسترس نمی باشند.

مثال: قسمت بدنه برای پکیج `cust_sal` را ایجاد کنید. همانطور که می بینید کد مربوط به زیربرنامه `find_sal` در این قسمت تعریف می شود.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
    PROCEDURE find_sal(c_id customers.id%TYPE) IS
    c_sal customers.salary%TYPE;
```

```
        BEGIN
                SELECT salary INTO c_sal
                FROM customers
                WHERE id = c_id;
                dbms_output.put_line('Salary: '|| c_sal);
        END find_sal;
END cust_sal;
/
```

Package body created.

13.3 چرا از پکیج استفاده می شود؟

1. با استفاده از پکیج، کد نویسی به روش ماژولار انجام می گیرد.

2. پنهان سازی جزییات پیاده سازی بدلیل ذخیره و تغییر کد برنامه در قسمت بدنه.

3.بهبود PERFORMANCE از آنجایی که با اولین دسترسی به پکیج تمام محتویات پکیج در حافظه فیزیکی قرار می گیرد.

4. می توان OBJECTهای مختلف (پروسیجر، تابع و ...) را در داخل یک پکیج تعریف کرد و یک GRANT کلی بر روی آن پکیج به کاربر مورد نظر داد.

13.4 استفاده از اجزای پکیج

اجزای پکیج از جمله توابع، پروسیجرها و متغیرها که در قسمت مشخصات پکیج عنوان شده اند با علامت . قابل دسترس و فراخوانی هستند:

```
package_name.element_name;
```

به این ترتیب **از متود (METHOD)** یک پکیج استفاده می شود.

مثال : از متود find_sal که در پکیج cust_sal تعریف شده است استفاده کنید.

```
 DECLARE
code customers.id%type := &cc_id;
BEGIN
cust_sal.find_sal(code);
END;
/
```

خروجی

```
Enter value for cc_id: 1
Salary: 3000
PL/SQL procedure successfully completed.
```

مثال 2: جدول customers را در نظر بگیرید:

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 3000.00 |
| 2 | Khilan | 25 | Delhi | 3000.00 |
| 3 | kaushik | 23 | Kota | 3000.00 |
| 4 | Chaitali | 25 | Mumbai | 7500.00 |
| 5 | Hardik | 27 | Bhopal | 9500.00 |
| 6 | Komal | 22 | MP | 5500.00 |
+----+----------+-----+-----------+----------+
```

قسمت مشخصات پکیج را تعریف کنید:

```
CREATE OR REPLACE PACKAGE c_package AS
-- Adds a customer
PROCEDURE addCustomer(c_id customers.id%type,
```

```
c_name customerS.name%type,
c_age customers.age%type,
c_addr customers.address%type,
c_sal customers.salary%type);
-- Removes a customer
PROCEDURE delCustomer(c_id customers.id%TYPE);
--Lists all customers
PROCEDURE listCustomer;
END c_package;
/
```

```
Package created.
```

قسمت بدنه پکیج را تعریف کنید:

```
CREATE OR REPLACE PACKAGE BODY c_package AS
PROCEDURE addCustomer(c_id customers.id%type,
c_name customerS.name%type,
c_age customers.age%type,
c_addr customers.address%type,
c_sal customers.salary%type)
IS
BEGIN
        INSERT INTO customers (id,name,age,address,salary)
        VALUES(c_id, c_name, c_age, c_addr, c_sal);
END addCustomer;
PROCEDURE delCustomer(c_id customers.id%type) IS
BEGIN
        DELETE FROM customers
        WHERE id = c_id;
END delCustomer;
PROCEDURE listCustomer IS
        CURSOR c_customers is
```

```
        SELECT name FROM customers;
        TYPE c_list is TABLE OF customerS.name%type;
        name_list c_list := c_list();
        counter integer :=0;
BEGIN
        FOR n IN c_customers LOOP
                counter := counter +1;
                name_list.extend;
                name_list(counter) := n.name;
                dbms_output.put_line('Customer(' ||counter||
                ')'||name_list(counter));
        END LOOP;
END listCustomer;
END c_package;
/
```

Package body created.

از متودهایی که در پکیج فوق تعریف شده است استفاده کنید.

```
DECLARE
code customers.id%type:= 8;
BEGIN
c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
c_package.listcustomer;
c_package.delcustomer(code);
c_package.listcustomer;
END;
/
```

خروجی

Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
PL/SQL procedure successfully completed

14 COLLECTION در اوراکل PL/SQL

در این فصل انواع COLLECTION در PL/SQL را توضیح می دهیم. COLLECTION یک مجموعه ترتیبی از اجزای مختلف است که نوع داده یکسان دارند. سه مدل COLLECTION قابل تعریف و استفاده هستند که یکی از آنها Varray یا آرایه است که در فصل ششم آن را توضیح دادیم. در جدول زیر انواع COLLECTION و ویژگی های آنها را می بینید.

نوع COLLECTION	تعداد اجزا	نوع زیرنویس (SUBSCRIPT)	DENSE یا SPARSE	محل ساخت
Associative array (or index-by table)	**نامحدود**	**رشته یا INTEGER**	**هردو**	فقط در بلاک برنامه
Nested table	**نامحدود**	**INTEGER**	**هردو**	در بلاک برنامه یا به عنوان یک OBJECT در سطح SCHEMA
Variablesize array (Varray)	**محدود**	**INTEGER**	**DENSE**	در بلاک برنامه یا به عنوان یک OBJECT در سطح SCHEMA

نکته: منظور از نوع زیرنویس یا SUBSCRIPT همان کلیدی است که با استفاده از آن می توان به اجزای COLLECTION دسترسی داشت.

نکته: نمی توان در COLLECTION های از نوع DENSE یک عنصر میانی را حذف نمود و فقط انتهای آن قابل تغییر می باشد بنابراین انعطاف پذیر نیستند.

14.1 چرا از COLLECTION استفاده می شود؟

- سبب ایجاد سهولت در کد نویسی می شود.

- زمانی که نیاز به پردازش اجزا با نوع داده یکسان است با استفاده از زیرنویس می توان به راحتی به تمام اجزا دسترسی یافت.

- می توان از متودهای مختلف (FIRST,COUNT,...) کمک گرفت.

- باعث PERFORMANCE بهتر در پردازش داده های موقتی برنامه ها می شود و نیاز به فراخوانی از دیتابیس را کمتر می کند.

نکته: COLLECTIONهای NESTED TABLE و INDEX BY TABLE ساختار یکسان دارند و سطرها یا اجزای آنها با استفاده از زیرنویس قابل دسترس هستند. در ادامه این نوع از جدول ها را توضیح می دهیم.

INDEX BY TABLE 14.2

در این دسته از COLLECTIONها زیرنویس ها UNIQUE هستند و می توانند از نوع INTEGER یا رشته باشند. در ادامه سینتکس ساخت INDEX BY TABLE را می بینید که در آن نوع داده اجزا و زیرنویس آنها مشخص می شود:

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY
subscript_type;
table_name type_name;
```

مثال: یک جدول به صورت INDEX BY TABLE بسازید که مقدارهای عددی به همراه یک رشته 20 کاراکتری برای هر کدام از آنها را ذخیره کند.

نکته: متودهای FIRST و NEXT به اجزای اولیه و بعدی COLLECTION اشاره می کنند.

```
DECLARE
TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
salary_list salary;
name VARCHAR2(20);
BEGIN
     -- adding elements to the table
     salary_list('Rajnish') := 62000;
     salary_list('Minakshi') := 75000;
     salary_list('Martin') := 100000;
     salary_list('James') := 78000;
     -- printing the table
     name := salary_list.FIRST;
     WHILE name IS NOT null LOOP
          dbms_output.put_line
          ('Salary of ' || name || ' is ' ||
          TO_CHAR(salary_list(name)));
          name := salary_list.NEXT(name);
     END LOOP;
END;
/
```

خروجی

```
Salary of James is 78000
Salary of Martin is 100000
Salary of Minakshi is 75000
Salary of Rajnish is 62000
PL/SQL procedure successfully completed.
```

نکته: نوع داده اجزای INDEX BY TABLE می تواند با کلمه کلیدی %ROWTYPE معادل نوع داده یک جدول دیگر در دیتابیس در نظر گرفته شود یا با کلمه کلیدی %TYPE معادل نوع داده یک ستون از جدول دیگر باشد.

مثال: از نوع داده جدول customers استفاده نمایید و بجای متود FIRST و NEXT از CURSOR استفاده کنید.

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+
```

```
DECLARE
CURSOR c_customers is
select name from customers;
TYPE c_list IS TABLE of customerS.name%type INDEX BY binary_integer;
name_list c_list;
counter integer :=0;
BEGIN
        FOR n IN c_customers LOOP
                counter := counter +1;
                 name_list(counter) := n.name;
                dbms_output.put_line('Customer('||counter||
                '):'||name_list(counter));
```

```
    END LOOP;
END;
/
```

خروجی

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
PL/SQL procedure successfully completed
```

NESTED TABLE 14.3

NESTED TABLE همانند یک آرایه است با این تفاوت که:

1. آرایه تعداد اجزای از قبل مشخص شده دارد ولی سایز یک NESTED TABLE می تواند به صورت پویا افزایش یابد.

2. آرایه به صورت DENSE است و ترتیب زیرنویس ها حفظ می شود ولی در NESTED TABLE می توان هرکدام از اجزای میانی را حذف نمود.

نکته: NESTED TABLEها برخلاف INDEX BY TABLEها می توانند در یک ستون از جدول های دیتابیس ذخیره شوند.

در ادامه سینتکس ساخت NESTED TABLE را می بینید.

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
table_name type_name;
```

مثال:

```
DECLARE
TYPE names_table IS TABLE OF VARCHAR2(10);
TYPE grades IS TABLE OF INTEGER;
names names_table;
marks grades;
total integer;
BEGIN
names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
marks:= grades(98, 97, 78, 87, 92);
total := names.count;
dbms_output.put_line('Total '|| total || ' Students');
FOR i IN 1 .. total LOOP
dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
end loop;
END;
/
```

خروجی

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92
PL/SQL procedure successfully completed.
```

نکته: نوع داده اجزای NESTED TABLE می تواند با کلمه کلیدی %ROWTYPE معادل نوع داده یک جدول دیگر در دیتابیس در نظر گرفته شود یا با کلمه کلیدی %TYPE معادل نوع داده یک ستون از جدول دیگر باشد.

مثال:

```
DECLARE
CURSOR c_customers is
SELECT name FROM customers;
TYPE c_list IS TABLE of customerS.name%type;
name_list c_list := c_list();
counter integer :=0;
BEGIN
        FOR n IN c_customers LOOP
                counter := counter +1;
                name_list.extend;
                name_list(counter) := n.name;
                dbms_output.put_line('Customer('||counter||'):'||name_list(coun
                ter));
        END LOOP;
END;
/
```

خروجی

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
PL/SQL procedure successfully completed.
```

14.4 متودهای COLLECTION

همانطور که در مثال های قبل ملاحظه کردید برای راحتی کار با COLLECTIONها در PL/SQL تعدادی متود در نظر گرفته شده است که می توان از آنها استفاده نمود. در جدول زیر فهرست متودها و دلیل استفاده از آنها را می بینید.

نام متود	دلیل استفاده
EXISTS(n)	اگر جزء nام وجود داشته باشد مقدار TRUE برگردانده می شود.
COUNT	تعداد اجزای کنونی یک COLLECTION را برمی گرداند.
LIMIT	حداکثر طول قابل تعریف برای یک COLLECTION را نشان می دهد.
FIRST	اگر زیرنویس از نوع عددی باشد اولین عدد مربوط به زیرنویس را برمی گرداند.
LAST	اگر زیرنویس از نوع عددی باشد آخرین عدد مربوط به زیرنویس را برمی گرداند.
PRIOR(n)	زیرنویس قبل از زیرنویس n را برمی گرداند.
NEXT(n)	زیرنویس بعد از زیرنویس n را برمی گرداند.
EXTEND	یک جزء NULL به COLLECTION اضافه می کند.
EXTEND(n)	به تعداد n جزء NULL به COLLECTION اضافه می کند.
EXTEND(n,i)	به تعداد n کپی از جزء iام به COLLECTION اضافه می کند.
TRIM	یک جزء از آخر COLLECTION حذف می کند.
TRIM(n)	به تعداد n جزء از آخر COLLECTION حذف می کند.
DELETE	تمام اجزای یک COLLECTION را حذف می کند. بنابراین count برابر با 0 می شود.
DELETE(n)	در COLLECTION های از نوع NESTED یا INDEX BY جزء nام یا جزئی که زیرنویس آن برابر با n است را حذف می کند.

DELETE(m,n)	در COLLECTION های از نوع NESTED یا INDEX BY اجزای بین m و n را پاک می کند.

14.5 EXCEPTION ها

در جدول زیر خطاهایی که در COLLECTIONها رخ می دهند را ملاحظه می کنید.

نوع EXCEPTION	زمان رخداد
COLLECTION_IS_NULL	می خواهیم روی یک COLLECTION که NULL است عملیات انجام دهیم
NO_DATA_FOUND	زیرنویس به یک جزئی که حذف شده است یا وجود ندارد می دهیم.
SUBSCRIPT_BEYOND_COUNT	از زیرنویس فراتر از اجزای یک COLLECTION استفاده می کنیم.
SUBSCRIPT_OUTSIDE_LIMIT	زیرنویس خارج از محدوده مجاز است
VALUE_ERROR	مقدار مشخص شده برای زیرنویس مثفاوت از نوع داده آن است.

15 پکیج های DBMS در دیتابیس اوراکل

در دیتابیس اوراکل تعدادی پکیج به صورت BUILT IN تعریف شده اند که به آنها پکیج های DBMS می گویند. این پکیج ها کاربردهای مختلفی دارند و می توان از آنها در زمان ساخت برنامه ها بهره برد.

15.1 چگونه پکیج های DBMS ایجاد می شوند؟

بعد از نصب نرم افزار اوراکل و در زمان ایجاد دیتابیس، پکیج های DBMS از طریق فایل اسکریپت catproc.sql ساخته می شوند. این فایل در مسیر زیر قرار دارد:

$ORACLE_HOME/rdbms/admin

این اسکریپت با فراخوانی اسکریپت های دیگر، پکیج های DBMS را ایجاد می کند. البته زمانی که از DBCA جهت ایجاد دیتابیس استفاده می شود به صورت اتوماتیک و ضمنی این اسکریپت ها اجرا می شوند. مالک پکیج های DBMS کاربر SYS می باشد.

15.2 پکیج های DBMS مهم

در جدول زیر تعدادی از پکیج های DBMS و کاربرد آنها را مشاهده می کنید.

نام پکیج	کاربرد
dbms_xmlgen	یک دستور SQL را به عنوان ورودی دریافت می کند و قالب آن را به XML تبدیل می کند.
dbms_xplan	explain plan مربوط به دستورات را در قالب خاص نمایش می دهد و به مانیتور کردن PERFORMANCE کمک می کند.
dbms_sql	یک واسط به منظور استفاده از DYNAMIC SQL جهت اجرای DDL و DML و بلاک های برنامه است
dbms_shared_pool	حافظه SHARED POOL را کنترل می کند. سایز OBJECTهای SHARED POOL را مشخص می کند و می تواند آنها را PIN یا UNPIN کند(در SHARED POOL بمانند یا خارج شوند)

با استفاده از زیربرنامه های این پکیج می توان تمام یا قسمتی از OBJECT از نوع LOB را دستکاری کرد یا مورد دسترسی قرار داد. برای مثال می توان بخشی از یک LOB را در LOB دیگر COPY نمود.	**dbms_lob**
شامل زیربرنامه های مربوط به ایجاد زمانبند و یا STOP و START کردن آن در دیتابیس اوراکل است.	**dbms_scheduler**

15.3 پکیج DBMS_OUTPUT

با استفاده از پکیج DBMS_OUTPUT می توان خروجی بلاک های PL/SQL، زیربرنامه ها و TRIGGERها را مشاهده کرد.

مثال: نام تمام جدول های کاربر دیتابیس را نمایش دهید.

```
BEGIN
        dbms_output.put_line (user || ' Tables in the database:');
        FOR t IN (SELECT table_name FROM user_tables)
        LOOP
                dbms_output.put_line(t.table_name);
        END LOOP;
END;
/
```

زیربرنامه های پکیج DBMS_OUTPUT را در جدول زیر مشاهده می کنید:

کاربرد	زیربرنامه
خروجی را غیرفعال می کند	**DBMS_OUTPUT.DISABLE**
خروجی را فعال می کند. اگر buffer_size ارسال نشود به صورت نامحدود در نظر گرفته می شود.	**DBMS_OUTPUT.ENABLE(buffer_size IN INTEGER DEFAULT 20000)**
یک خط از اطلاعات بافر شده را نمایش می دهد.	**DBMS_OUTPUT.GET_LINE (line OUT VARCHAR2, status OUT INTEGER)**
آرایه ای از خطوط را از بافر نمایش می دهد.	**DBMS_OUTPUT.GET_LINES (lines OUT CHARARR, numlines IN OUT INTEGER)**
یک خط جدید نمایش می دهد.	**DBMS_OUTPUT.NEW_LINE;**
ورودی را یک خط جدید در بافر قرار می دهد.	**DBMS_OUTPUT.PUT_LINE(item IN VARCHAR2);**

مثال:

```
DECLARE
        lines dbms_output.chararr;
        num_lines number;
BEGIN
        -- enable the buffer with default size 20000
        dbms_output.enable;
        dbms_output.put_line('Hello Reader!');
        dbms_output.put_line('Hope you have enjoyed the tutorials!');
        dbms_output.put_line('Have a great time exploring pl/sql!');
        num_lines := 3;
        dbms_output.get_lines(lines, num_lines);
        FOR i IN 1..num_lines LOOP
                dbms_output.put_line(lines(i));
        END LOOP;
END;
/
```

خروجی

```
Hello Reader!
Hope you have enjoyed the tutorials!
Have a great time exploring pl/sql!
PL/SQL procedure successfully completed.
```

16 برنامه نویسی شی گرا در اوراکل PL/SQL

در PL/SQL با استفاده از نوع OBJECT (OBJECT TYPE) می توان برنامه نویسی شی گرا یا OBJECT ORIENTED انجام داد. درOBJECT واسط ها و جزییات پیاده سازی از هم جدا می شوند و در دیتابیس ذخیره گردند. این روش، زمان و هزینه پیاده سازی برنامه های پیچیده را کاهش می دهد و خاصیت انتزاع (ABSTRACTION) و ترکیبی بودن را برقرار می کند.

خاصیت ترکیبی:

نوع OBJECT همانند نوع داده رکورد، ترکیبی از عناصر مختلف است با این تفاوت که عملیات قابل انجام روی نوع OBJECT از پیش تعریف شده نیست و توسط کاربر تعریف می شود.

خاصیت انتزاع:

یک OBJECT از لحاظ مخفی بودن جزییات، همانند یک پروسیجر می ماند ولی نمی توان در بلاک برنامه PL/SQL، نوع OBJECT تعریف نمود بلکه می بایست در دیتابیس اوراکل تعریف و ذخیره شود.

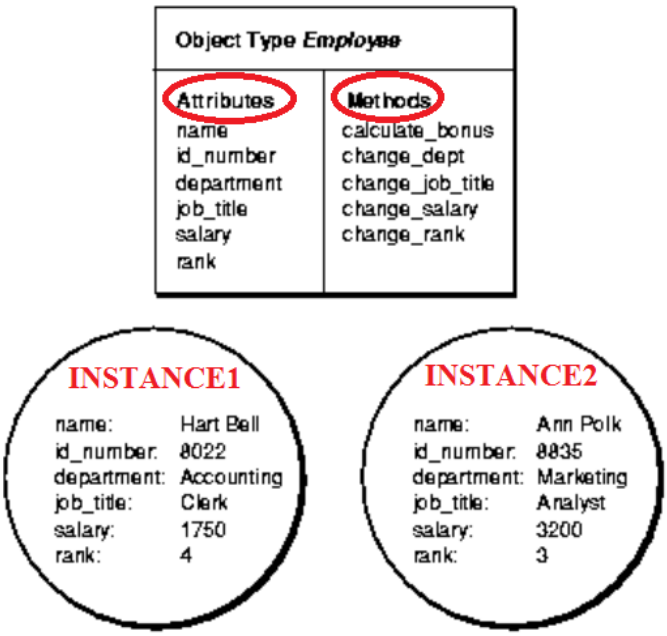
16.1 نوع OBJECT چیست؟

نوع OBJECT مجموعه ای از نوع داده ها با ساختار مختلف به همراه تابع و پروسیجر است. به متغییرهایی که ساختار داده در OBJECT را تشکیل می دهند ATTRIBUTE می گویند و توابع و پروسیجرها که خصوصیات رفتاری یک OBJECT را مشخص می کنند METHOD نامیده می شوند. برای مثال اگر انسان را یک OBJECT در نظر بگیریم، سن، وزن و جنسیت او به عنوان ATTRIBUTE و خوردن، آشامیدن و خوابیدن به عنوان METHOD درنظر گرفته می شوند.

فرض کنید نوع OBJECT کارمندان با ATTRIBUTE و METHODهای مختلف تعریف و در دیتابیس ذخیره می شود. بنابراین گروه های مختلف کارمندان بر اساس مشخصات و عملیات مورد نیاز فقط قسمتی از ویژگی های این OBJECT را استفاده خواهند کرد.

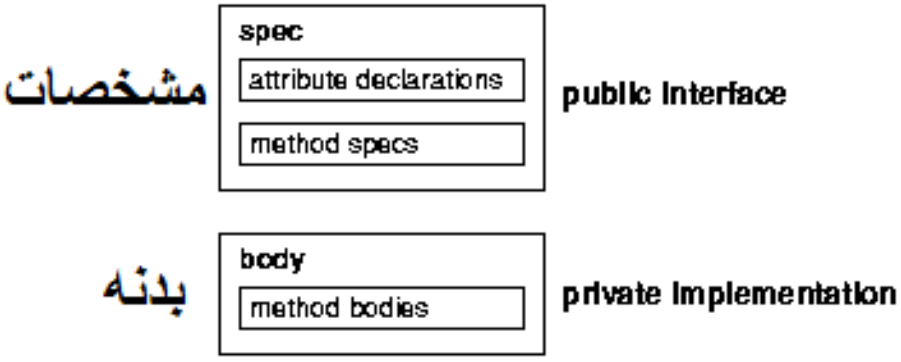
بعد از تعریف و ذخیره سازی نوع OBJECT در دیتابیس اوراکل، در زمان اجرای برنامه می توان به تعداد دلخواه INSTANCE از روی آن OBJECT ساخت و از آنها استفاده نمود.

مثال: ساختن دو INSTANCE از نوع OBJECT کارمندان.



16.2 ساختار OBJECT

یک OBJECT همانند پکیج از دو قسمت مشخصات (SPEC) و بدنه تشکیل شده است. در قسمت SPEC واسط های PUBLIC تعریف می شوند که شامل ATTRIBUTEها و نام METHODهایی است که عملیات مختلف روی داده ها انجام می دهند. در قسمت بدنه نیز METHODها (پروسیجر و تابع) تعریف می شوند. بنابراین تمام اطلاعاتی که در اختیار برنامه کاربران قرار می گیرد در SPEC تعریف شده است.



مثال:

```
CREATE  OR REPLACE TYPE Complex AS OBJECT (
  rpart REAL,  -- attribute
  ipart REAL,
  MEMBER FUNCTION plus (x Complex) RETURN Complex,  -- method
  MEMBER FUNCTION less (x Complex) RETURN Complex,
  MEMBER FUNCTION times (x Complex) RETURN Complex,
  MEMBER FUNCTION divby (x Complex) RETURN Complex
);

CREATE TYPE BODY Complex AS
  MEMBER FUNCTION plus (x Complex) RETURN Complex IS
  BEGIN
    RETURN Complex(rpart + x.rpart, ipart + x.ipart);
  END plus;

  MEMBER FUNCTION less (x Complex) RETURN Complex IS
  BEGIN
    RETURN Complex(rpart - x.rpart, ipart - x.ipart);
  END less;

  MEMBER FUNCTION times (x Complex) RETURN Complex IS
  BEGIN
    RETURN Complex(rpart * x.rpart - ipart * x.ipart,
                   rpart * x.ipart + ipart * x.rpart);
  END times;

  MEMBER FUNCTION divby (x Complex) RETURN Complex IS
    z REAL := x.rpart**2 + x.ipart**2;
  BEGIN
    RETURN Complex((rpart * x.rpart + ipart * x.ipart) / z,
                   (ipart * x.rpart - rpart * x.ipart) / z);
  END divby;
END;
```

نکته: در یک OBJECT:

-نمی توان CONSTANT، CURSOR ، EXCEPTION و TYPE تعریف و استفاده نمود.

-حداقل باید یک ATTRIBUTE تعریف شود ولی می توان حداکثر تا 1000 مورد تعریف کرد.

-استفاده از METHOD و قسمت بدنه اختیاری است.

نکته: هر ATTRIBTE همانند یک متغییر باید نام واحد در حوزه آن OBJECT داشته باشد و نوع داده آن می تواند از نوع های مختلف باشد به غیر از:

-LONG و LONGROW

-ROWID و UROWID

-BINARY_INTEGER، BOOLEAN، RECORD ، REF_CURSOR ، %TYPE و %ROWTYPE

-نوع داده هایی که در داخل یک پکیج تعریف شده اند.

نکته: در زمان تعریف ATTRIBUTE نمی توان از مقدار پیش فرض یا initialize و NULL استفاده نمود.

16.3 دسترسی به METHODها و ATTRIBUTEهای OBJECT

برای دسترسی به اجزای OBJECT ابتدا باید یک INSTANCE از آن OBJECT ساخته شود و مقدارهای مناسب به اجزا اختصاص داده شود. سپس با استفاده از نام آن INSTANCE و علامت **.** می توان به METHODها یا ATTRIBUTEهای آن OBJECT دسترسی داشت.

مثال:

```
CREATE OR REPLACE TYPE address AS OBJECT
(house_no varchar2(10),
street varchar2(30),
city varchar2(20),
state varchar2(10),
pincode varchar2(10)
```

```
);
/

DECLARE
residence address;
BEGIN
residence := address('103A', 'M.G.Road', 'Jaipur', 'Rajasthan','201301');
dbms_output.put_line('House No: '|| residence.house_no);
dbms_output.put_line('Street: '|| residence.street);
dbms_output.put_line('City: '|| residence.city);
dbms_output.put_line('State: '|| residence.state);
dbms_output.put_line('Pincode: '|| residence.pincode);
END;
/
```

خروجی:

```
House No: 103A
Street: M.G.Road
City: Jaipur
State: Rajasthan
Pincode: 201301
PL/SQL procedure successfully completed.
```

نکته: METHOD هایی که به منظور دستکاری ATTRIBUTEهای یک OBJECT استفاده می شوند MEMBER نامیده می شوند.

نکته: توابعی که یک OBJECT جدید برمی گردانند CONSTRUCTOR نامیده می شوند. هر OBJECT یک CONSTRUCTOR از پیش تعریف شده دارد که نام آن برابر با نام OBJECT است. در مثال قبل نام تابع CONSTRUCTOR برابر با address() است:

```
residence := address('103A', 'M.G.Road', 'Jaipur', 'Rajasthan','201301');
```

16.4 METHODهای مقایسه ای

با استفاده از METHODهای مقایسه ای می توان OBJECTهای مختلف را به روش دلخواه مقایسه کرد. METHODهای مقایسه ای عبارتند از:

1.MAP METHOD:

فرض کنید یک OBJECT برای چهارضلعی ها تعریف شده است. اگر بخواهیم اندازه دو NSTANCE از این OBJECT را مقایسه کنیم باید با استفاده از تابع MAP مساحت هر INSTANCE را با INSTANCE دیگر مقایسه کنیم.

مثال: برای چهارضلعی یک OBJECT تعریف می کنیم:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
width number,
member function enlarge( inc number) return rectangle,
member procedure display,
map member function measure return number
);
/
```

در قسمت بدنه تابع های MAP و MEMBER تعریف می شود.

```
CREATE OR REPLACE TYPE BODY rectangle AS
MEMBER FUNCTION enlarge(inc number) return rectangle IS
BEGIN
return rectangle(self.length + inc, self.width + inc);
END enlarge;
MEMBER PROCEDURE display IS
BEGIN

dbms_output.put_line('Length: '|| length);
dbms_output.put_line('Width: '|| width);
END display;
MAP MEMBER FUNCTION measure return number IS
BEGIN
return (sqrt(length*length + width*width));
```

```
END measure;
END;
/
```

ایجاد INSTANCE و استفاده از METHODها:

```
DECLARE
r1 rectangle;
r2 rectangle;
r3 rectangle;
inc_factor number := 5;
BEGIN
r1 := rectangle(3, 4);
r2 := rectangle(5, 7);
r3 := r1.enlarge(inc_factor);
r3.display;
IF (r1 > r2) THEN -- calling measure function
r1.display;
ELSE
r2.display;
END IF;
END;
/
```

در این مثال با استفاده از تابع MEMBER به نام enlarge مشخصات INSTANCE شماره 3 بر اساس INSTANCE شماره 1 تعریف می شود و در زمان مقایسه بین INSTANCE شماره 1 با شماره 2 از تابع MAP استفاده می شود.

خروجی:

```
Length: 8
Width: 9
Length: 5
Width: 7
PL/SQL procedure successfully completed.
```

:ORDER METHOD.2

در تابع ORDER می توان از یکسری منطق های داخلی نیز استفاده نمود تا دو INSTANCE را مقایسه کنیم.

مثال: برای چهارضلعی یک OBJECT تعریف می کنیم.

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
width number,
member procedure display,
order member function measure(r rectangle) return number
);
/
```

در قسمت بدنه، توابه MEMBER و ORDER رل تعریف می کنیم:

```
CREATE OR REPLACE TYPE BODY rectangle AS
MEMBER PROCEDURE display IS
BEGIN
dbms_output.put_line('Length: '|| length);
dbms_output.put_line('Width: '|| width);
END display;
ORDER MEMBER FUNCTION measure(r rectangle) return number IS
BEGIN
IF(sqrt(self.length*self.length + self.width*self.width)>
sqrt(r.length*r.length + r.width*r.width)) then
return(1);
 ELSE
return(-1);
END IF;
END measure;
END;
/
```

ایجاد INSTANCE و مقایسه آنها:

```
DECLARE
r1 rectangle;
r2 rectangle;
```

```
BEGIN
r1 := rectangle(23, 44);
r2 := rectangle(15, 17);
r1.display;
r2.display;
IF (r1 > r2) THEN -- calling measure function
r1.display;
ELSE
r2.display;
END IF;
END;
/
```

خروجی

```
Length: 23
Width: 44
Length: 15
Width: 17
Length: 23
Width: 44
```

نکته: در توابع MEMBER می توان از یک پارامتر داخلی به نام SELF استفاده نمود که اشاره به همان INSTANCE که تابع MEMBER از آن فراخوانی شده است می کند. در مثال قبل از SELF استفاده شده است.

16.5 وراثت در OBJECTها

می توان از روی OBJECTهایی که وجود دارند یک OBJECT فرزند ایجاد کرد به شرطی که OBJECT اصلی به صورت NOT FINAL تعریف شده باشد.

مثال: ایجاد OBJECT جهارضلعی به صورت NOT FINAL:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
width number,
```

```
member function enlarge( inc number) return rectangle,
NOT FINAL member procedure display) NOT FINAL
/
```

قسمت بدنه:

```
CREATE OR REPLACE TYPE BODY rectangle AS
MEMBER FUNCTION enlarge(inc number) return rectangle IS
BEGIN
return rectangle(self.length + inc, self.width + inc);
END enlarge;
MEMBER PROCEDURE display IS
BEGIN
dbms_output.put_line('Length: '|| length);
dbms_output.put_line('Width: '|| width);
END display;
END;
/
```

تعریف یک OBJECT فرزند به نام tabletop :

```
CREATE OR REPLACE TYPE tabletop UNDER rectangle
(
material varchar2(20),
OVERRIDING member procedure display
)
/
```

قسمت بدنه:

```
CREATE OR REPLACE TYPE BODY tabletop AS
OVERRIDING MEMBER PROCEDURE display IS
BEGIN
dbms_output.put_line('Length: '|| length);
dbms_output.put_line('Width: '|| width);
dbms_output.put_line('Material: '|| material);
END display;
/
```

ایجاد INSTANCE و استفاده از tabletop:

```
DECLARE
t1 tabletop;
t2 tabletop;
BEGIN
t1:= tabletop(20, 10, 'Wood');
t2 := tabletop(50, 30, 'Steel');
t1.display;
t2.display;
END;
/
```

خروجی:

```
Length: 20
Width: 10
Material: Wood
Length: 50
Width: 30
Material: Steel
PL/SQL procedure successfully completed.
```

نکته: اگر یک OBJECT با عبارت NOT INSTANTIABLE تعریف شود، نمی توان INSTANCE از روی آن ساخت بلکه باید برای آن OBJECT فرزند تعریف کرد و از آن استفاده نمود.

مثال: تعریف OBJECT به صورت NOT INSTANTIABLE:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
width number,
NOT INSTANTIABLE NOT FINAL MEMBER PROCEDURE display)
NOT INSTANTIABLE NOT FINAL
/
```

نکته: توجه شود که در برخی متون به INSTANCE که از روی OBJECT ساخته می شود OBJECT نیز می گویند.