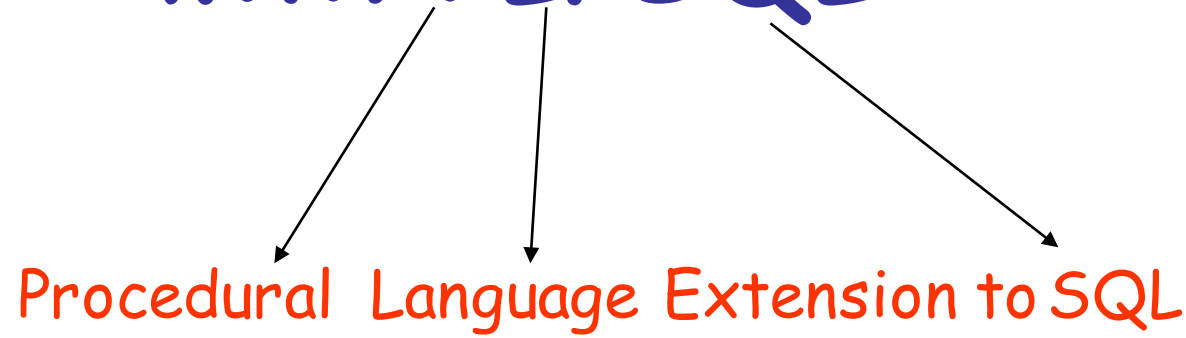
Programming in Oracle with PL/SQL



Database Languages

Procedural Language & Non-Procedural Language

- In a procedural language **each step of the task (i.e.** procedure **) is defined** precisely .
- **Procedural languages** are used in the **traditional programming** that is based on algorithms or a logical step-by-step process for solving a problem.
- **Examples: C,**C++, Java.
- **WHAT & HOW** a process should be done !!

- **Non-procedural**
  Programming languages allow users and professional programmers to **specify the results they want without specifying** how to solve the problem.
- **Ex. SQL (**Structured Query Language) used for RDBMS.
- **Non-procedural language** is concerned with the
- **WHAT not the HOW.**

SQL

- Though SQL is the natural language of the DBA it suffers from various inherent disadvantages, as compared to conventional Programming Languages.
- It does not have procedural capabilities such as looping, conditional checking, branching.

What is PL/SQL

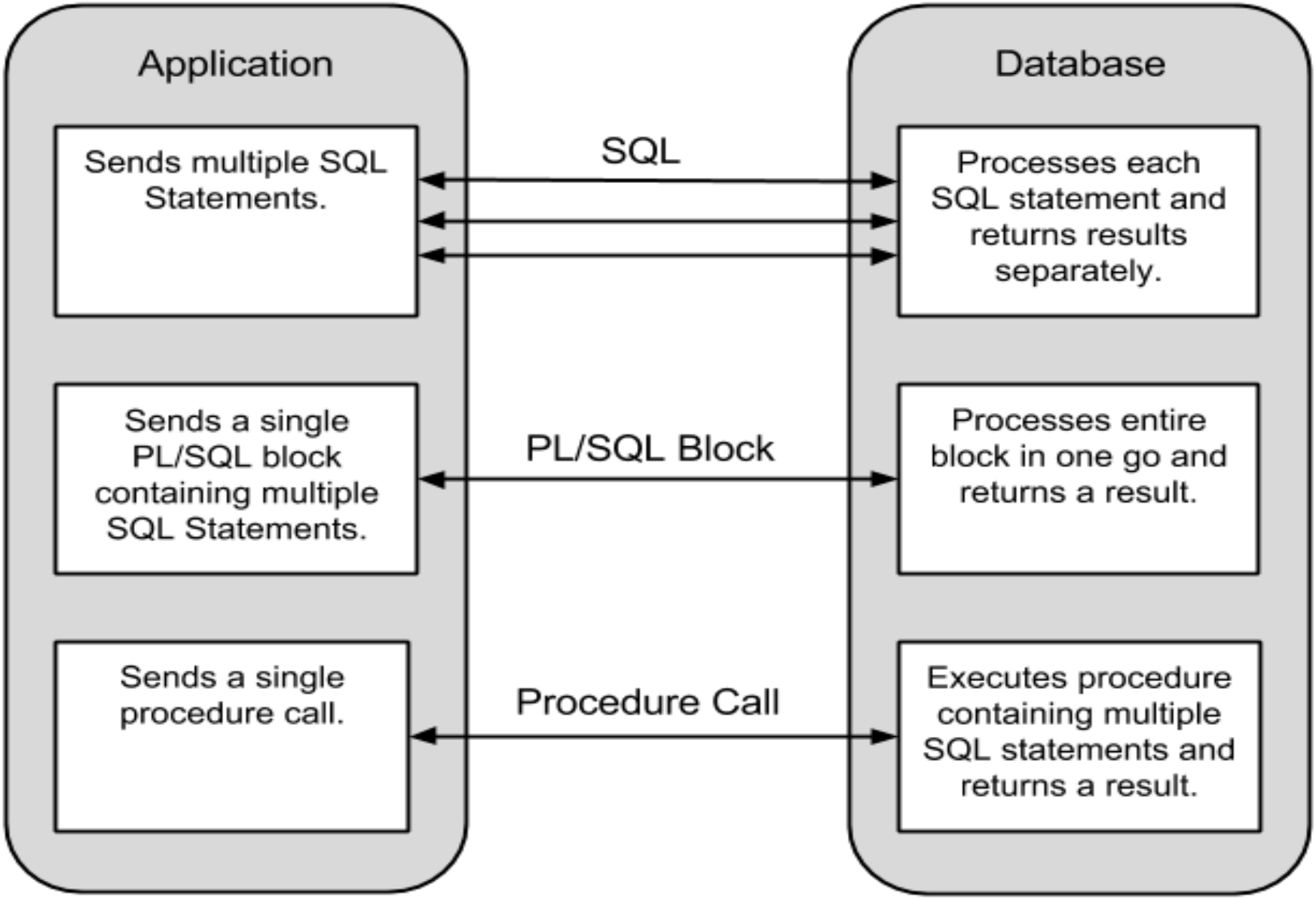
- Procedural Language – SQL
- An extension to SQL with design features of programming languages (procedural and object oriented)
- PL/SQL and Java are both supported as internal host languages within Oracle products.

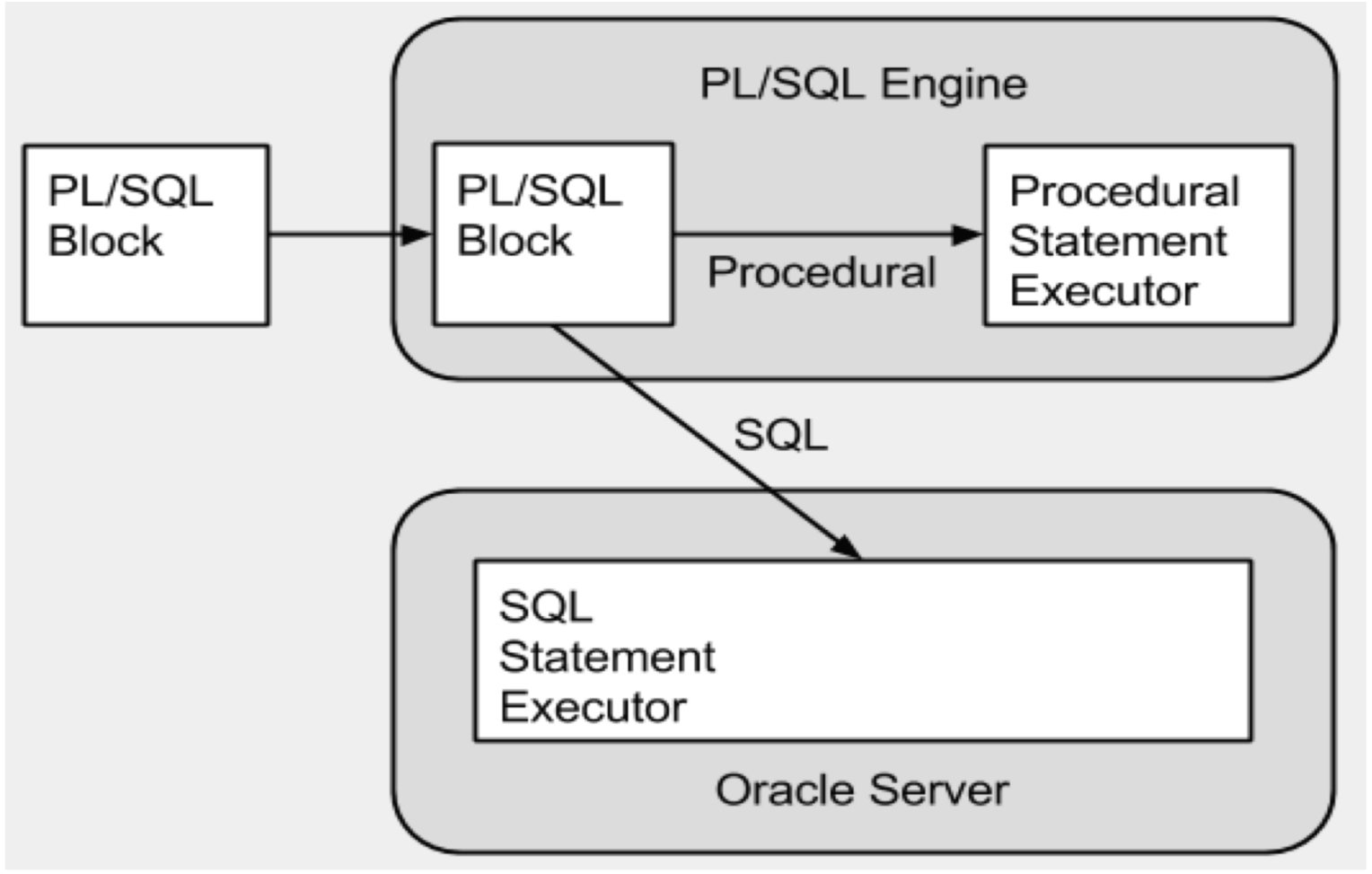
Why PL/SQL

- Acts as host language for stored procedures and triggers.
- Provides the ability to add middle tier business logic to client/server applications.
- Provides Portability of code from one environment to another
- Improves performance of multi-query transactions.
- Provides error handling

What is so great about PL/SQL anyway?

- PL/SQL is a procedural extension of SQL
- Making it extremely simple to write procedural code that includes SQL as if it were a single language.
- The data types in PL/SQL are a super-set of those in the database, so you rarely need to perform data type conversions when using PL/SQL.
- Average Java or .NET programmer how they find handling date values coming from a database. They can only wish for the simplicity of PL/SQL.
- When coding business logic in middle tier applications, a single business transaction may be made up of multiple interactions between the application server and the database.
- This adds a significant overhead associated with network traffic.
- In comparison, building all the business logic as PL/SQL in the database means client code needs only a single database call per transaction, reducing the network overhead significantly.


Application
Database
Sends multiple SQL Statements.
SQL
Processes each SQL statement and returns results separately.
Sends a single PL/SQL block containing multiple SQL Statements.
PL/SQL Block
Processes entire block in one go and returns a result.
Sends a single procedure call.
Procedure Call
Executes procedure containing multiple SQL statements and returns a result.


PL/SQL Engine
PL/SQL Block
PL/SQL Block
Procedural
Procedural Statement Executor
SQL
SQL Statement Executor
Oracle Server

Advantages of PL/SQL

- **These are the Advantages of PL/SQL**
- ***Block Structures*:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.
- ***Procedural Language Capability*:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).
- ***Better Performance*:** PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.
- ***Error Handling*:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

PL/SQL

- Features
  - Tight integration with SQL
    - Supports data types, functions, pseudo-columns, etc.
  - Increased performance
    - A block of statements sent as a single statement
  - Increased productivity
    - Same techniques can be used with most Oracle products
  - Portability
    - Works on any Oracle platform
  - Tighter security
    - Users may access database objects without granted privileges

PL/SQL

- Allows using general programming tools with SQL, for example: loops, conditions, functions, etc.
- This allows a lot more freedom than general SQL, and is lighter-weight than JDBC.
- We write PL/SQL code in a regular file, for example PL.sql, and load it with @PL in the sqlplus console.

PL/SQL Blocks

- PL/SQL code is built of Blocks, with a unique structure.
- There are two types of blocks in PL/SQL:

1. **Anonymous Blocks**: have no name (like scripts)
   - can be written and executed immediately in SQLPLUS
   - can be used in a trigger
2. **Named Blocks**:
   - Procedures
   - Functions

Anonymous Block Structure:

```
DECLARE          (optional)
   /* Here you declare the variables you will use in this
      block */
BEGIN            (mandatory)
   /* Here you define the executable statements (what
      the block DOES!)*/
EXCEPTION        (optional)
   /* Here you define the actions that take place if an
      exception is thrown during the run of this block */
END;             (mandatory)
/
```

Always put a new line with only a / at the end of a block! (This tells Oracle to run the block)

A correct completion of a block will generate the following message:

PL/SQL procedure successfully completed

PL/SQL Block Types

Anonymous

```
DECLARE
BEGIN
    -statements
EXCEPTION
END;
```

Procedure

```
PROCEDURE <name>
IS
BEGIN
-statements
EXCEPTION
END;
```

Function

```
FUNCTION <name>
RETURN <datatype>
IS
BEGIN
-statements
EXCEPTION
END;
```

PL/SQL Variables

- Variables are local to the code block
- Names can be up to 30 characters long and must begin with a character
- Declaration is like that in a table
  - Name then data type the semi-colon
  - Can be initialized using := operator in the declaration
  - Can be changed with := in the begin section
  - Can use constraints
- Variables can be composite or collection types
  - Multiple values of different or same type

PL/SQL Variable Types

- Scalar (char, varchar2, number, date, etc)
- Composite (%rowtype)
- Reference (pointers)
- LOB (large objects)

Note: Non PL/SQL variables include bind variables, host ("global") variables, and parameters.

Variable Naming Conventions

- Two variables can have the same name if they are in different blocks (bad idea)
- The variable name should not be the same as any table column names used in the block.

PL/SQL is strongly typed

- All variables must be declared before their use.
- The assignment statement

  : =

  is not the same as the equality operator

  =

- All statements end with a ;

Common PL/SQL Data Types

- CHAR (max_length)
- VARCHAR2 (max_length)
- NUMBER (precision, scale)
- INTEGER
- RAW (max_length)
- DATE
- BOOLEAN (true, false, null)
- Also LONG, LONG RAW and LOB types but the capacity is usually less in PL/SQL than SQL

PL/SQL Variable Constraints

- NOT NULL
  - Can not be empty
- CONSTANT
  - Can not be changed

PL/SQL Variables Examples

Age number;

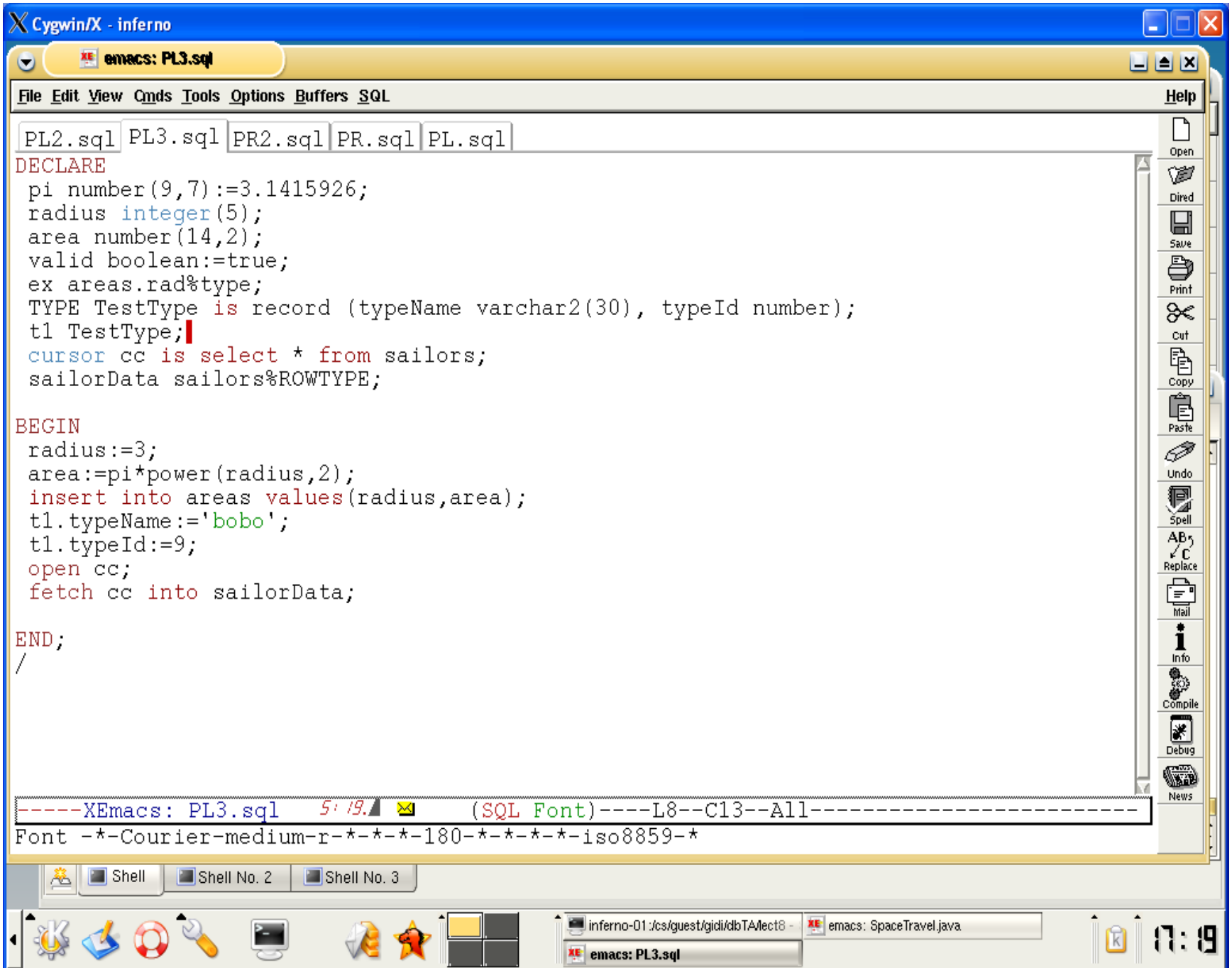
Last char (10);

DVal Date := Sysdate;

SID number not null;

Adjust constant number := 1;

CanLoop boolean := true


Cygwin/X - inferno
emacs: PL3.sql
File Edit View Cmds Tools Options Buffers SQL
Help
PL2.sql
PL3.sql
PR2.sql
PR.sql
PL.sql
DECLARE
pi number(9,7):=3.1415926;
radius integer(5);
area number(14,2);
valid boolean:=true;
ex areas.rad%type;
TYPE TestType is record (typeName varchar2(30), typeId number);
t1 TestType;
cursor cc is select * from sailors;
sailorData sailors%ROWTYPE;
BEGIN
radius:=3;
area:=pi*power(radius,2);
insert into areas values(radius,area);
t1.typeName:='bobo';
t1.typeId:=9;
open cc;
fetch cc into sailorData;
END;
/
Open
Dired
Save
Print
Cut
Copy
Paste
Undo
Spell
Replace
Mail
Info
Compile
Debug
News
-----XEmacs: PL3.sql 5:19 (SQL Font)----L8--C13--All----------
Font -*-Courier-medium-r-*-*-*-180-*-*-*-*-iso8859-*
Shell
Shell No. 2
Shell No. 3
inferno-01:/cs/guest/gidi/dbTA/lect8 -
emacs: SpaceTravel.java
emacs: PL3.sql
17:19

DECLARE

Syntax

```
identifier [CONSTANT] datatype [NOT NULL]
    [:= | DEFAULT expr];
```

Examples

Notice that PL/SQL includes all SQL types, and more...

```
Declare
  birthday    DATE;
  age         NUMBER(2) NOT NULL := 27;
  name        VARCHAR2(13) := 'Levi';
  magic       CONSTANT NUMBER := 77;
  valid       BOOLEAN NOT NULL := TRUE;
```

Declaring Variables with the %TYPE Attribute

Examples

Accessing column sname in table Sailors

```
DECLARE
   sname              Sailors.sname%TYPE;
   fav_boat           VARCHAR2(30);
   my_fav_boat        fav_boat%TYPE  := 'Pinta';
...
```

Accessing another variable

Declaring Variables with the %ROWTYPE Attribute

Declare a variable with the type of a ROW of a table.

Accessing table Reserves

```
reserves_record    Reserves%ROWTYPE;
```

And how do we access the fields in reserves_record?

```
reserves_record.sid:=9;
Reserves_record.bid:=877;
```

Creating a PL/SQL Record

A record is a type of variable which we can define (like 'struct' in C or 'object' in Java)

```
DECLARE
   TYPE sailor_record_type  IS RECORD
      (sname       VARCHAR2(10),
       sid         VARCHAR2(9),
       age         NUMBER(3),
      rating        NUMBER(3));
   sailor_record    sailor_record_type;
...
BEGIN
   Sailor_record.sname:='peter';
   Sailor_record.age:=45;
...
```

CURSORS

- A cursor is a private set of records
- An Oracle Cursor = VB recordset = JDBC ResultSet
- Implicit cursors are created for every query made in Oracle
- Explicit cursors can be declared by a programmer within PL/SQL.

Creating a Cursor

- We create a Cursor when we want to go over a result of a query (like ResultSet in JDBC)
- Syntax Example:

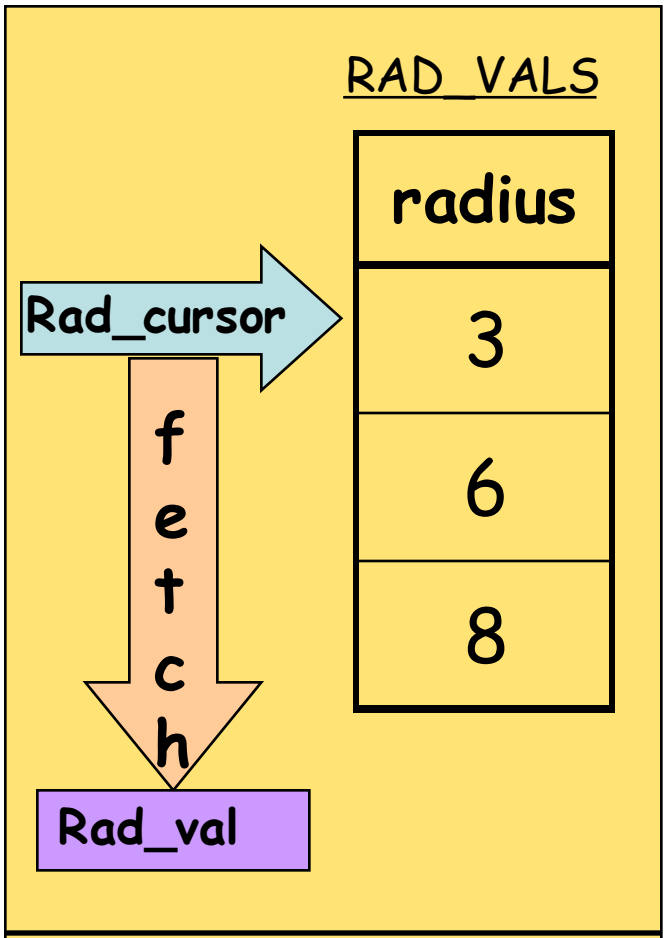
```
DECLARE
   cursor c is select * from sailors;
   sailorData sailors%ROWTYPE;

BEGIN
   open c;
   fetch c into sailorData;
```

sailorData is a variable that can hold a ROW from the sailors table

Here the first row of sailors is inserted into sailorData

Example




```
DECLARE
   Pi constant NUMBER(8,7) := 3.1415926;
  area NUMBER(14,2);
  cursor rad_cursor is select * from RAD_VALS;
  rad_value rad_cursor%ROWTYPE;

BEGIN
   open rad_cursor;
  fetch rad_cursor into rad_val;
  area:=pi*power(rad_val.radius,2);
  insert into AREAS values (rad_val.radius, area);
  close rad_cursor;
END;
/
```

AREAS

Radius	Area
3	28.27

1

```
DECLARE
  ...
  cursor rad_cursor is select * from
RAD_VALS;
  rad_value rad_cursor%ROWTYPE;
BEGIN
   open rad_cursor;
  fetch rad_cursor into rad_value;
  area:=pi*power(rad_value.radius,2);
  insert into AREAS values
(rad_value.radius,
  area);
  ...
```

2

```
DECLARE
  ...
  cursor rad_cursor is select * from
RAD_VALS;
  rad_value RAD_VALS%ROWTYPE;
BEGIN
   open rad_cursor;
  fetch rad_cursor into rad_value;
  area:=pi*power(rad_value.radius,2);
  insert into AREAS values (rad_value.radius,
area);
  ...
```

3

```
DECLARE
  ...
  cursor rad_cursor is select radius from

RAD_VALS;
  rad_value RAD_VALS.radius%TYPE;
BEGIN
   open rad_cursor;
  fetch rad_cursor into rad_value;
  area:=pi*power(rad_value,2);
  insert into AREAS values (rad_value, area);
  ...
```

4

```
DECLARE
  ...
  cursor rad_cursor is select *  from
RAD_VALS;
  rad_value RAD_VALS.radius%TYPE;
BEGIN
   open rad_cursor;
  fetch rad_cursor into rad_value;
  area:=pi*power(rad_value,2);
  insert into AREAS values (rad_value, area);
  ...
```

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open.
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row.
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far.

SELECT Statements

```
DECLARE
   v_sname   VARCHAR2(10);
   v_rating  NUMBER(3);
BEGIN
   SELECT   sname, rating
     INTO   v_sname, v_rating
     FROM   Sailors
    WHERE   sid = '112';
END;
/
```

- INTO clause is required.
- Query must return exactly one row.
- Otherwise, a NO_DATA_FOUND or TOO_MANY_ROWS exception is thrown

Conditional logic

Condition:

```
If <cond>
   then <command>
elsif <cond2>
   then <command2>
else
      <command3>
end if;
```

Nested conditions:

```
If <cond>
   then
      if <cond2>
         then
            <command1>
       end if;
 else <command2>
 end if;
```

IF-THEN-ELSIF Statements

```
. . .
IF rating > 7 THEN
   v_message := 'You are great';
ELSIF rating >= 5 THEN
   v_message := 'Not bad';
ELSE
   v_message := 'Pretty bad';
END IF;
. . .
```

Suppose we have the following table:

```
create table mylog(
      who varchar2(30),
      logon_num number
);
```

mylog

who	logon_num
Peter	3
John	4
Moshe	2

- Want to keep track of how many times someone logged on to the DB
- When running, if user is already in table, increment logon_num. Otherwise, insert user into table

Solution

```
DECLARE
   cnt  NUMBER;
BEGIN
   select count(*)
    into cnt
    from mylog
   where who = user;

   if cnt > 0 then
     update mylog
       set logon_num = logon_num + 1
     where who = user;
   else
     insert into mylog values(user, 1);
   end if;
   commit;
end;
/
```

SQL Cursor

SQL cursor is automatically created after each SQL query. It has 4 useful attributes:

SQL%ROWCOUNT	**Number of rows affected by the most recent SQL statement (an integer value).**
SQL%FOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows.**
SQL%NOTFOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows.**
SQL%ISOPEN	**Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.**

Solution (2)

```
BEGIN
 update mylog
   set logon_num = logon_num + 1
   where who = user;

 if SQL%ROWCOUNT = 0 then
   insert into mylog values(user, 1);
 end if;
 commit;
END;
/
```

Loops: Simple Loop

```
create table number_table(
      num NUMBER(10)
);
```

```
DECLARE
   i  number_table.num%TYPE  := 1;
BEGIN
   LOOP
      INSERT INTO number_table
        VALUES(i);
      i := i + 1;
      EXIT WHEN i > 10;
   END LOOP;
END;
```

Loops: Simple Cursor Loop

```
create table number_table(
      num NUMBER(10)
);
```

```
DECLARE
   cursor c is select * from number_table;
   cVal c%ROWTYPE;
BEGIN
   open c;
   LOOP
      fetch c into cVal;
      EXIT WHEN c%NOTFOUND;
      insert into doubles values(cVal.num*2);
   END LOOP;
END;
```

Loops: FOR Loop

```
DECLARE
   i        number_table.num%TYPE;
BEGIN
   FOR i IN 1..10 LOOP
      INSERT INTO number_table  VALUES(i);
   END LOOP;
END;
```

Notice that i is incremented automatically

Loops: For Cursor Loops

```
DECLARE
   cursor c is select * from number_table;

BEGIN
    for num_row in c loop
        insert into doubles_table
                        values(num_row.num*2);
    end loop;
END;
/
```

Notice that a lot is being done implicitly: declaration of num_row, open cursor, fetch cursor, the exit condition

Loops: WHILE Loop

```
DECLARE
TEN number:=10;
i       number_table.num%TYPE:=1;
BEGIN
   WHILE i <= TEN LOOP
       INSERT INTO number_table
       VALUES(i);
       i := i + 1;
   END LOOP;
END;
```

Printing Output

- You need to use a function in the DBMS_OUTPUT package in order to print to the output
- If you want to see the output on the screen, you must type the following (before starting):

```
set serveroutput on format wrapped (word_wrapped,
   truncated) size 1000000
```

- Then print using
  - dbms_output. put_line(*your_string*);
  - dbms_output.put(*your_string*);

Input and output example

```
set serveroutput on format wrapped size 1000000
ACCEPT high PROMPT 'Enter a number: '

DECLARE
i  number_table.num%TYPE:=1;
BEGIN
  dbms_output.put_line('Look Ma, I can print from PL/SQL!!!');
  WHILE i <= &high LOOP
     INSERT INTO number_table
     VALUES(i);
     i := i + 1;
  END LOOP;
END;
```

Accept Syntax:
ACC[EPT] *variable* [NUM[BER] | CHAR | DATE | BINARY_FLOAT | BINARY_DOUBLE] [FOR[MAT] *format*] [DEF[AULT] *default*] [PROMPT *text*|NOPR[OMPT]] [HIDE]

Example Program

PL/SQL program to display name and salary of each employee in department

```
DECLARE CURSOR emp_cur
IS SELECT ename, sal FROM emp
WHERE deptno = 10 ORDER BY sal DESC; BEGIN
FOR emp_rec IN emp_cur LOOP DBMS_OUTPUT.PUT_LINE
('Employee ' || emp_rec.ename || ' earns ' || TO_CHAR
(emp_rec.sal) || ' dollars.'); END LOOP;
END;
/
```

Output:

```
Employee KING earns 5000 dollars.
Employee SCOTT earns 3000 dollars.
Employee JONES earns 2975 dollars.
Employee ADAMS earns 1100 dollars.
Employee JAMES earns 950 dollars.
```

Write PL/SQL Block Which Will Display Reverse Number Of Given Number.

```
SQL> DECLARE
  2    no integer;
  3    rev integer:=0;
  4  BEGIN
  5    no:=&'Enter_No.';
  6    WHILE (no>0)  LOOP
  7    rev:=rev*10+MOD(no,10);
  8    no:=TRUNC(no/10);
  9    END LOOP;
 10     DBMS_OUTPUT.PUT_LINE('Reverse No. Is'||' : '||rev);
 11  END;
 12  /
Enter value for enter_no: 123456
old   5:    no:='&Enter_No.';
new   5:    no:='123456';
Reverse No. Is : 654321
```

Write PL/SQL Block To Display Factorial Of Given Number.

```
DECLARE
   num NUMBER:='&Input_Number';
   i NUMBER;
   f NUMBER:=1;
BEGIN
   FOR i IN 1..num LOOP
      f := f * i;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE(f||' Is Factorial Of '||num);
END;
/
Enter value for input_number:  5

120 Is Factorial Of 5
```

Write a Pl/SQL code block that will accept an account number from user and debit an amount of Rs 2000 from the account has a minimum balance of 500 after the amount is debited .The process is to be fired on the account table.

```
Declare
  Acct_balance number(11,2);
  Acct_no varchar2(6);
  Debit_amt number(5):=2000
  Min_bal constant number(5,2):=500.00
Begin
/* Accept an account _no from user*/
Acct_no:=&acct_no;
/* retreiving bal.from account table where account_no in the table is equal to the account_no enetered
   by the user*/
SELECT bal INTO acct_balance
FROM accounts
WHERE account_id=acct_no;
/*substract if the balance is greater than equal to the min bal. of Rs 500 .If the condition is satisfied an
   amount of Rs 2000 is subtracted from the balance of the corresponding account no.*/
IF acct_balance >+MIN_BAL THEN
UPDATE account SET bal=bal-debit_amt
WHERE account_id=acct_no;
ENDIF;
END
```

- Write a Pl/SQL block of code that **first inserts a record in'Emp table'.**
  - **Update the salaries of Blake and Clarke by Rs 2000 &Rs 1200.**
  - **Then check to see that the total salary dos not exceed 20000.**
  - **If the total salary is greater than 20000 then undo the updates made to the salaries of Blake & Clarke.**

```
DECLARE
 Total_sal  number (9);
 BEGIN
 INSERT INTO emp values (‘E005’.’John’,1000);
 /*Defining a savepoint */
 SAVEPOINT no_updates;
 /*updation of the salaries of Blake and Clarke in the ‘Emp table’*/
 UPDATE emp SET sal=sal+2000
  where emp_name=‘Blake;’;
 UPDATE emp SET sal =sal + 1500
 Where emp_name =‘clark’
SELECT sum(sal) INTO total_sal
FROM emp;
 IF total_sal > 20000 THEN
ROLLBACK To Savepoint no_update;
 ENDIF;
COMMIT;
END;
```

Reminder- structure of a block

```
DECLARE          (optional)
    /* Here you declare the variables you will use in this
       block */
BEGIN            (mandatory)
    /* Here you define the executable statements (what
       the block DOES!)*/
EXCEPTION        (optional)
    /* Here you define the actions that take place if an
       exception is thrown during the run of this block */
END;             (mandatory)
/
```

Exceptions

Syntax of exception handler:

EXCEPTION WHEN *ex_name_1* THEN *statements_1* -- Exception handler
WHEN *ex_name_2* OR *ex_name_3* THEN *statements_2* -- Exception handler
WHEN OTHERS THEN *statements_3* -- Exception handler
END;

Exception Categories:

Internally defined: The runtime system raises internally defined exceptions implicitly (automatically).

Examples: ORA-00060 (deadlock detected while waiting for resource) and ORA-27102 (out of memory).

Exception

Predefined: an internally defined exception that PL/SQL has given a name.

Example: ORA-06500 (PL/SQL: storage error) has the predefined name STORAGE_ERROR.

User-defined: You can declare your own exceptions in the declarative part of any PL/SQL anonymous block, subprogram, or package.

Example: you might declare an exception named insufficient_funds to flag overdrawn bank accounts.

You must raise user-defined exceptions explicitly.

Exception

Category	Definer	Has Error Code	Has Name	Raised Implicitly	Raised Explicitly
Internally defined	Runtime system	Always	Only if you assign one	Yes	Optionally
Predefined	Runtime system	Always	Always	Yes	Optionally
User-defined	User	Only if you assign one	Always	No	Always

Naming Internally Defined Exceptions

```
DECLARE deadlock_detected EXCEPTION;
PRAGMA
EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
…
EXCEPTION WHEN deadlock_detected THEN
…
END;
/
```

Trapping Exceptions

- Here we define the actions that should happen when an exception is thrown.
- Example Exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - ZERO_DIVIDE
- When handling an exception, consider performing a rollback

```
DECLARE
    num_row  number_table%ROWTYPE;
BEGIN
    select *
    into num_row
    from number_table;
    dbms_output.put_line(1/num_row.num);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
      dbms_output.put_line('No data!');
    WHEN TOO_MANY_ROWS THEN
      dbms_output.put_line('Too many!');
    WHEN OTHERS THEN
      dbms_output.put_line(‘Error’);
end;
```

User-Defined Exception

```
DECLARE
    e_number1   EXCEPTION;
    cnt         NUMBER;
BEGIN
    select count(*)
    into cnt
    from number_table;

    IF cnt = 1 THEN RAISE e_number1;
    ELSE dbms_output.put_line(cnt);
    END IF;

EXCEPTION
    WHEN e_number1 THEN
      dbms_output.put_line('Count = 1');
end;
```

Predefined Exceptions

- **INVALID_NUMBER (ORA-01722)**
  - **Attempted to store non-numeric data in a variable with a numeric data type**
- **NO_DATA_FOUND (ORA-01403)**
  - **Query resulted in no rows being found**
- **NOT_LOGGED_ON (ORA-01012)**
  - **Not currently connected to an Oracle database**
- **TOO_MANY_ROWS (ORA-01422)**
  - **A SELECT INTO statement returned more than one row**

Predefined Exceptions (cont.)

- **DUP_VALUE_ON_INDEX (ORA-00001)**
  - **Value inserted for a primary key is not unique**
- **VALUE_ERROR (ORA-06502)**
  - **The value being placed in a variable is the wrong length or data type**
- **ZERO_DIVIDE (ORA-01476)**
  - **An attempt was made to divide a number by zero**

Functions and Procedures

- Up until now, our code was in an anonymous block
- It was run immediately
- It is useful to put code in a function or procedure so it can be called several times
- Once we create a procedure or function in a Database, it will remain until deleted (like a table).

Stored Procedures

```
CREATE PROCEDURE ProcedureName(parameter1 datatype, parameter2 datatype, ...) AS
   Additional declarations of local variables
BEGIN
   executable section
EXCEPTION
   Optional exception section
END;
```

Stored Procedures

- The first line is called the Procedure Specification
- The remainder is the Procedure Body
- A procedure is compiled and loaded in the database as an object
- Procedures can have parameters passed to them

Stored Procedures

- Run a procedure with the PL/SQL EXECUTE command
- Parameters are enclosed in parentheses

Stored Functions

- Like a procedure except they return a single value

PARAMETERS

- Parameters are the means to pass values to and from the calling environment to the server.
- These are the values that will be processed or returned via the execution of the procedure.
- There are three types of parameters:
  IN, OUT, and IN OUT.
- Modes specify whether the parameter passed is read in or a receptacle for what comes out.

Types of Parameters

Mode	Description	Usage
IN	Passes a value into the program	Read only value Constants, literals, expressions Cannot be changed within program Default mode
OUT	Passes a value back from the program	Write only value Cannot assign default values Has to be a variable Value assigned only if the program is successful
IN OUT	Passes values in and also send values back	Has to be a variable Value will be read and then written

FORMAL AND ACTUAL PARAMETERS

Formal parameters are the names specified within parentheses as part of the header of a module.

Actual parameters are the values—expressions specified within parentheses as a parameter list—when a call is made to the module.

The formal parameter and the related actual parameter must be of the same or compatible data types.

MATCHING ACTUAL AND FORMAL PARAMETERS

Two methods can be used to match actual and formal parameters: positional notation and named notation.

Positional notation is simply association by position: The order of the parameters used when executing the procedure matches the order in the procedure's header exactly.

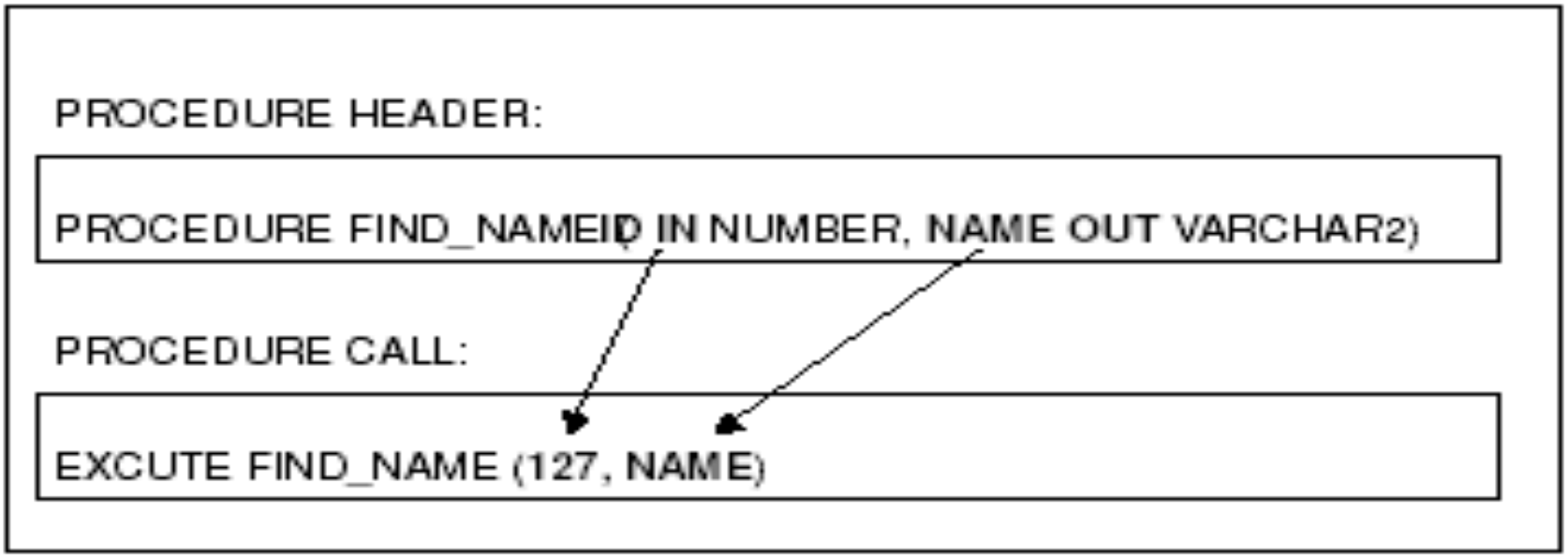
Named notation is explicit association using the symbol =>

```
Syntax: formal_parameter_name =>
argument_value
```

In named notation, the order does not matter.

If you mix notation, list positional notation before named notation.

MATCHING ACTUAL AND FORMAL PARAMETERS



Formal and Actual Parameters

```
DECLARE emp_num NUMBER(6) := 120; bonus NUMBER(6) := 100;
merit NUMBER(4) := 50;
PROCEDURE raise_salary (
emp_id NUMBER, -- formal parameter
amount NUMBER -- formal parameter
) IS
BEGIN
UPDATE employees
SET salary = salary + amount
WHERE employee_id = emp_id;
END raise_salary;
BEGIN
raise_salary(emp_num, bonus); -- actual parameters
raise_salary(emp_num, merit + bonus); -- actual parameters
END;
/
```

Subprogram Calls using Positional Named and Mixed Notation

```
SQL> DECLARE
2 emp_num NUMBER(6) := 120;
3 bonus NUMBER(6) := 50;
4 PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
5 BEGIN
6 UPDATE employees SET salary =
7 salary + amount WHERE employee_id = emp_id;
8 END raise_salary;
9 BEGIN
10 -- Positional notation:
11 raise_salary(emp_num, bonus);
12 -- Named notation (parameter order is insignificant):
13 raise_salary(amount => bonus, emp_id => emp_num);
14 raise_salary(emp_id => emp_num, amount => bonus);
15 -- Mixed notation:
16 raise_salary(emp_num, amount => bonus);
17 END;
18 /
```

Function Call Using Position, Named and Mixed Notation

```
SQL> CREATE OR REPLACE FUNCTION compute_bonus (emp_id NUMBER, bonus
NUMBER)
2 RETURN NUMBER
3 IS
4 emp_sal NUMBER;
5 BEGIN
6 SELECT salary INTO emp_sal
7 FROM employees
8 WHERE employee_id = emp_id;
9 RETURN emp_sal + bonus;
10 END compute_bonus;
11 / Function created.

SQL> SELECT compute_bonus(120, 50) FROM DUAL; -- positional
2 SELECT compute_bonus(bonus => 50, emp_id => 120) FROM DUAL; --
named
3 SELECT compute_bonus(120, bonus => 50) FROM DUAL; -- mixed
4 SQL>
```

Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE
procedure_name
  [(parameter1 [mode1] datatype1,
   parameter2 [mode2] datatype2,
   . . .)]
IS|AS
PL/SQL Block;
```

- Modes:
  - IN: procedure must be called with a value for the parameter. Value cannot be changed
  - OUT: procedure must be called with a variable for the parameter. Changes to the parameter are seen by the user (i.e., call by reference)
  - IN OUT: value can be sent, and changes to the parameter are seen by the user
- Default Mode is: IN

Example- what does this do?

Table mylog

who	logon_num
Pete	3
John	4
Joe	2

```
create or replace procedure
num_logged
(person IN mylog.who%TYPE,
 num OUT mylog.logon_num%TYPE)
IS
BEGIN
     select logon_num
     into num
     from mylog
     where who = person;
END;
/
```

Calling the Procedure

```
declare
     howmany  mylog.logon_num%TYPE;
begin
     num_logged(‘John',howmany);
     dbms_output.put_line(howmany);
end;
/
```

Errors in a Procedure

- When creating the procedure, if there are errors in its definition, they will not be shown
- To see the errors of a procedure called *myProcedure*, type

  SHOW ERRORS PROCEDURE *myProcedure*

  in the SQLPLUS prompt
- For functions, type

  SHOW ERRORS FUNCTION *myFunction*

FUNCTIONS

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL block that *returns* a single value.
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.
- The datatype of the return value must be declared in the header of the function.
- A function is not a stand-alone executable in the way that a procedure is: It must be used in some context. You can think of it as a sentence fragment.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

Creating a Function

- Almost exactly like creating a procedure, but you supply a return type

```
CREATE [OR REPLACE] FUNCTION
function_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

A Function

```
create or replace function
rating_message(rating IN NUMBER)
return VARCHAR2
AS
BEGIN
  IF rating > 7 THEN
    return 'You are great';
  ELSIF rating >= 5 THEN
    return  'Not bad';
  ELSE
    return 'Pretty bad';
  END IF;
END;
/
```

NOTE THAT YOU DON'T SPECIFY THE SIZE

Calling the function

```
declare
    paulRate:=9;
Begin
dbms_output.put_line(ratingMessage(paulRate));
end;
/
```

Creating a function:

```
create or replace function squareFunc(num in number)
return number
is
BEGIN
return num*num;
End;
/
```

Using the function:

```
BEGIN
dbms_output.put_line(squareFunc(3.5));
END;
/
```

FUNCTIONS

The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.

The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).

A function may have IN, OUT, or IN OUT parameters. but you rarely see anything except IN parameters.

Example

```
CREATE OR REPLACE FUNCTION show_description
(i_course_no number)
RETURN varchar2
AS
v_description varchar2(50);
BEGIN
SELECT description
INTO v_description
FROM course
WHERE course_no = i_course_no;
RETURN v_description;
EXCEPTION
WHEN NO_DATA_FOUND
THEN
RETURN('The Course is not in the database');
WHEN OTHERS
THEN
RETURN('Error in running show_description');
END;
```

Making Use Of Functions

- **In a anonymous block**

```
SET SERVEROUTPUT ON
DECLARE
v_description VARCHAR2(50);
BEGIN
v_description :=
show_description(&sv_cnumber);
DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

- **In a SQL statement**

```
SELECT course_no, show_description(course_no)
FROM course;
```

Packages

- Functions, Procedures, Variables can be put together in a package
- In a package, you can allow some of the members to be "public" and some to be "private"
- There are also many predefined Oracle packages
- Won't discuss packages in this course

Triggers

- Triggers are special procedures which we want activated when someone has performed some action on the DB.
- For example, we might define a trigger that is executed when someone attempts to insert a row into a table, and the trigger checks that the inserted data is valid.
- To be continued…