Programming in Oracle with PL/SQL

Overview

- Overview of PL/SQL
- Data type and Variables
- Program Structures
- Triggers
- Database Access Using Cursors
- Records
- PL/SQL Tables
- Built-in Packages
- Error-Handling
- PL/SQL Access to Oracle 10g Objects

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.

Other Databases

- All have procedural facilities
- SQL is not functionally complete
  - Lacks full facilities of a programming language
- So top up functionality by embedding SQL in a procedural language
- PL/SQL techniques are specific to Oracle
  - but procedures and functions can be ported to other systems

Why use PL/SQL

- Manage business rules – through *middle layer* application logic.
- Generate code for triggers
- Generate code for interface
- Enable database-centric client/server applications

Using PL/SQL as a programming language

- Permits all operations of standard programming languages e.g.
  - Conditions IF-THEN-ELSE-END IF;
  - Jumps GOTO
- Provides loops for controlling iteration
  - LOOP-EXIT; WHEN-END LOOP; FOR-END LOOP; WHILE-END LOOP
- Allows extraction of data into variables and its subsequent manipulation

Overview

- Overview of PL/SQL
- Data type and Variables
- Program Structures
- Triggers
- Database Access Using Cursors
- Records
- PL/SQL Tables
- Built-in Packages
- Error-Handling
- PL/SQL Access to Oracle 10g Objects

Use of Data-Types

<variable-name> <datatype> [not null][: =<initial-value>];

<constant-name> constant <datatype> : = <value>];

- Number – used to store any number
- Char(size) & varchar2(size) e.g.: char(10) – used to store alphanumerical text strings, the char data type will pad the value stored to the full length declared.
- Date – used to store dates
- Long – used to store large blocks of text up to 2 gigabytes in length (limited operations)

More data-types

- Long raw – stores large blocks of data stored in binary format
- Raw – stores smaller blocks of data in binary formal
- Rowid – used to store the special format of rowid's on the database

Variable and constant declaration

<variable-name> <datatype> [not null][: =<initial-value>];

<constant-name> constant <datatype> [: = <value>];

```
DECLARE
        i BINARY_INTEGER;
        cno NUMBER(5) NOT NULL := 1111;
        cname VARCHAR2(30);
        commission REAL(5,2) := 12.5;
        MAXCOLUMNS CONSTANT INTEGER(2) := 30;
        hired_data DATE;
        done BOOLEAN;
```

Anchored Data Type

<variable-name> <object> %type [not null][: =<initial-value>];

- Variables can also be declared to have anchored data types
- Data types **are determined by** looking up another object's data type.
- This another data type could be a column in the database, thereby providing the ability to match the data types of PL/SQL variables with the data types of columns defined in the database.

Anchored Data Type Example

<variable-name> <object> %type [not null][: =<initial-value>];

```
commission real(5,2) := 12.5

X commission%type;

Cname employee.empname%type;
```

- Record.element notation will address components of tuples *(dot notation)*

employee

Empid	empname	addr1	addr2	addr3	postcode	grade	salary

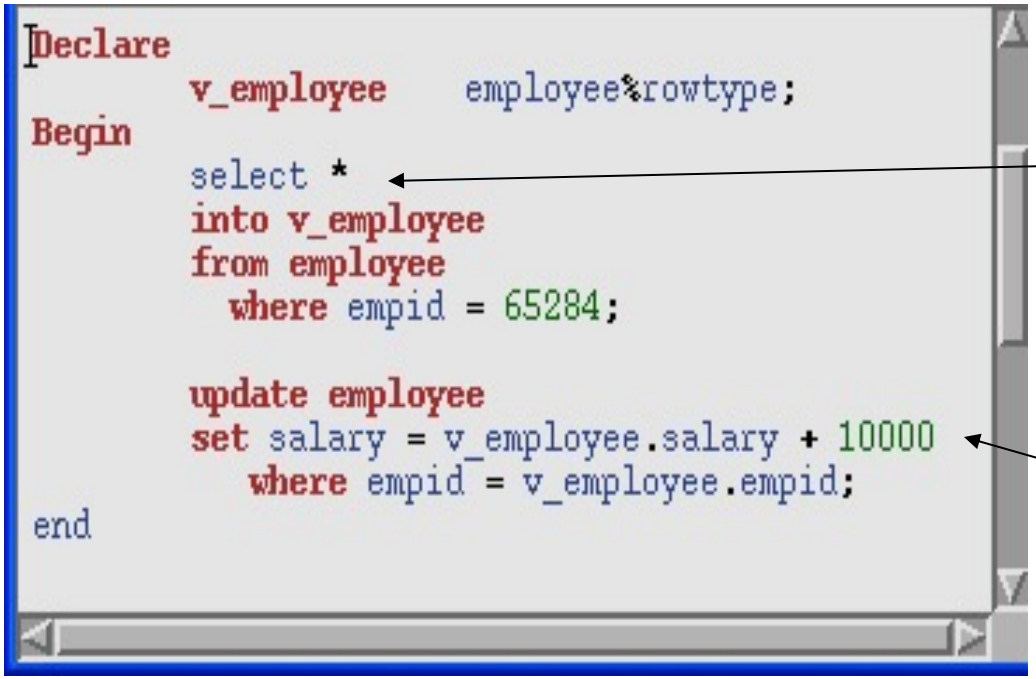
Anchored Data Type Example

- Select values into PL/SQL variables
  - using INTO
- %rowtype allows full rows to be selected into one variable

V_employee employee%rowtype

Empid	empname	addr1	addr2	addr3	postcode	grade	salary

Anchored Data Type Example



```
Declare
        v_employee    employee%rowtype;
Begin
        select *
        into v_employee
        from employee
          where empid = 65284;

        update employee
        set salary = v_employee.salary + 10000
           where empid = v_employee.empid;
end
```

Selects entire row of data into 1 variable called v_employee

Is updating the value of salary based on selected element of a variable

p1.sql

Overview

- Overview of PL/SQL
- Data type and Variables
- Program Structures
- Triggers
- Database Access Using Cursors
- Records
- PL/SQL Tables
- Built-in Packages
- Error-Handling
- PL/SQL Access to Oracle 10g Objects

Program Structures: Procedures and Functions

- A set of SQL and PL/SQL statements grouped together as a unit (*block*) to solve a specific problem or perform a set of related tasks.
- An *anonymous block* is a PL/SQL block that appears within your application and it is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear.
- A *stored procedure* is a PL/SQL block that Oracle stores in the database and can be called by name from an application. May or may not return a value.
- Functions always return a single value to the caller; procedures do not return values to the caller.
- Packages are groups of procedures and functions.

PL/SQL Blocks

- PL/SQL code is built of Blocks, with a unique structure.
- **Anonymous Blocks:** have no name (like scripts)
  - can be written and executed immediately in SQLPLUS
  - can be used in a trigger

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)

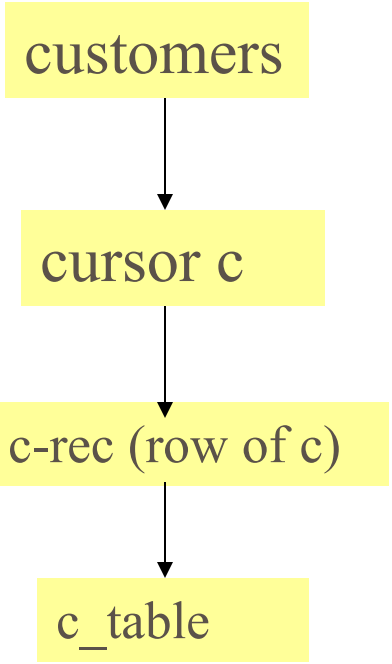
A correct completion of a block will generate the following message:

PL/SQL procedure successfully completed

Anonymous Blocks

```
DECLARE
  TYPE customers_table IS TABLE  OF customers%ROWTYPE
       INDEX BY BINARY_INTEGER;

  c_table customers_table;
  CURSOR c IS select cno,cname,street,zip,phone
              from   customers;
  c_rec c%ROWTYPE;
  i BINARY_INTEGER;
BEGIN
  i := 1;
  FOR c_rec in c LOOP
    EXIT WHEN c%NOTFOUND;
    c_table(i) := c_rec;
    i := i + 1;
  END LOOP;
  for i in 1..c_table.count loop
    DBMS_OUTPUT.PUT_LINE('c_table(' || i || ').cname = ' || c_table(i).cname);
  end loop;
END;
/
```



SQL> start p2.sql

Gets all the rows from customers table and prints the names of the customers on the screen. It uses tables and cursors.

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;
...
```

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

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
);
```

- 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

mylog

who	logon_num
Peter	3
John	4
Moshe	2

Solution

```
DECLARE
  cnt  NUMBER;
BEGIN
  select logon_num
   into cnt     //variable store current logon nums
   from mylog
  where who = user;//func returns current user name

  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 number_table 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 (refer to slide 19 for details)

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 size 1000000

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

Input and output example

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

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

Functions and Procedures

Functions and Procedures

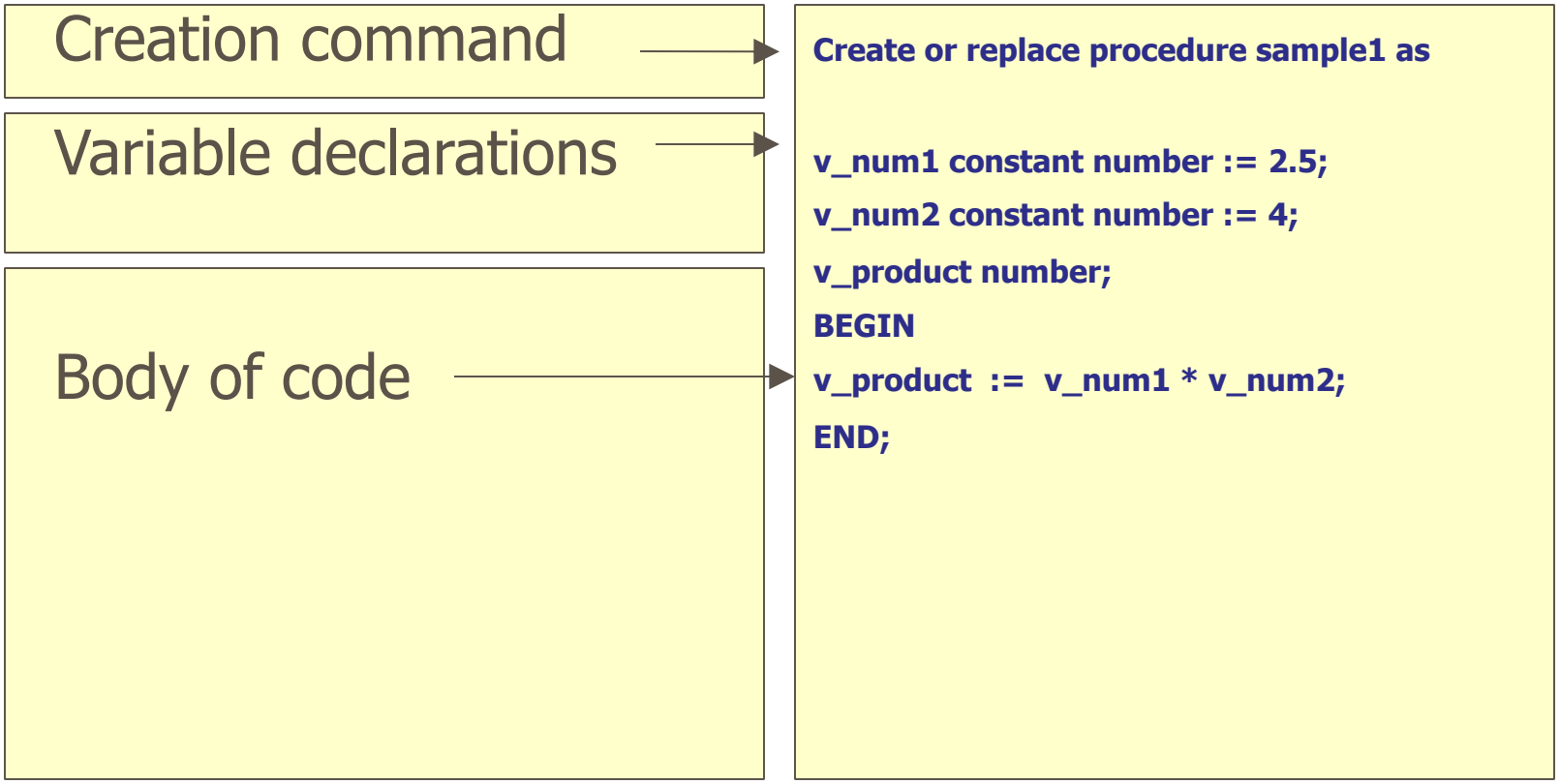
- It is useful to put code in a function or procedure so it can be called several times
- Once we create a procedure or function in a Database, it will remain until deleted (like a table).

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

Procedures


Creation command
Variable declarations
Body of code
Create or replace procedure sample1 as
v_num1 constant number := 2.5;
v_num2 constant number := 4;
v_product number;
BEGIN
v_product := v_num1 * v_num2;
END;

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;
/
```

More procedures: p3.sql

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*

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;
/
```

More functions: p4.sql

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;
/
```

Stored Procedures and Functions

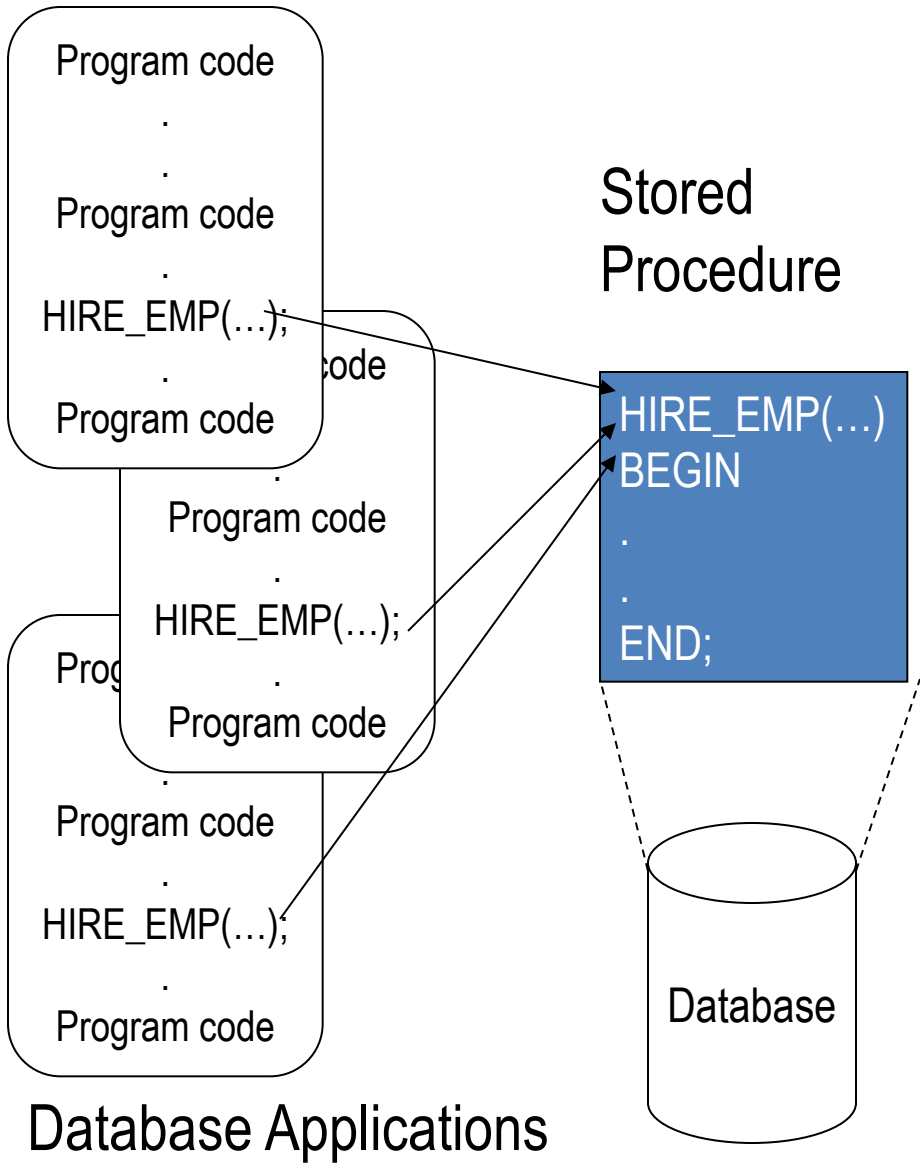
- The procedures and functions we discussed were called from within the executable section of the anonymous block.
- It is possible to store the procedure or function definition in the database and have it invoked from various of environments.
- This feature allows for sharing of PL/SQL code by different applications running in different places.

Stored Procedures

Created in a user's schema and stored centrally, in *compiled form* in the database as a named object that can be:

- interactively executed by a user using a tool like SQL*Plus
- called explicitly in the code of a database application, such as an Oracle Forms or a Pre compiler application, or in the code of another procedure or trigger

When PL/SQL is stored in the database, applications can send blocks of PL/SQL to the database rather than individual SQL statements → reducing network traffic. .



Stored Procedures and Functions

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

- AS keyword means stored procedure/function
- IS keyword means part of anonymous block
- So does stored function

Stored function: p5.sql

p2.sql | p5.sql

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

create or replace function get_city(
      cnum in customers.cno%type)
    return zipcodes.city%type as

  ccity zipcodes.city%type;

begin
  select city
  into   ccity
  from   customers,zipcodes
  where  cno = cnum and
         customers.zip = zipcodes.zip;
  return (ccity);
end;
/
```

Call Stored function

```
SQL>SELECT CNO, CNAME, get_city(cno)
2 from customers;

CNO        CNAME    GET_CITY(CNO)
------     --------  ------------------
1111       Charles    Wichita
2222       Bertram   Wichita
```

get_city function returns city name given customer number.

customers(cno, cname, zip) zipcodes(cnum, zip, city)

Benefits of Stored Procedures I

- Security
  - Control data access through procedures and functions.
  - E.g. grant users access to a procedure that updates a table, but not grant them access to the table itself.
- Performance

  The information is sent only once between database and application and thereafter invoked when it is used.
  - Network traffic is reduced compared with issuing individual SQL statements or sending the text of an entire PL/SQL block
  - A procedure's compiled form is readily available in the database, so no compilation is required at execution time.
  - The procedure might be cached

Benefits of Procedures II

- Memory Allocation
  - Stored procedures take advantage of the shared memory capabilities of Oracle
  - Only a single copy of the procedure needs to be loaded into memory for execution by multiple users.
- Productivity
  - By designing applications around a common set of procedures, you can avoid redundant coding and increase your productivity.
  - Procedures can be written to insert, update, or delete rows from a table and then called by any application without rewriting the SQL statements necessary to accomplish these tasks.
  - If the methods of data management change, only the procedures need to be modified, not all of the applications that use the procedures.

Benefits of Procedures III

- Integrity
  - Stored procedures improve the integrity and consistency of your applications. By developing all of your applications around a common group of procedures, you can reduce the likelihood of committing coding errors.
  - You can test a procedure or function to guarantee that it returns an accurate result and, once it is verified, reuse it in any number of applications without testing it again.
  - If the data structures referenced by the procedure are altered in any way, only the procedure needs to be recompiled; applications that call the procedure do not necessarily require any modifications.

Packages

- collection of procedures and function
- 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

Packages Example

- A package called process_orders in p6.sql
- Contains three procedures
  - add_order takes user input and insert a new row to orders table.
  - add_order_details receives input and add a new row to odetails table.
  - ship_order updates shipped value for the order.

Execute procedures in the package:

```
SQL> execute process_orders.add_order(2000,111,1000,null);

SQL> execute process_orders.add_order_details(2000,10509,50) ;

SQL> execute process_orders.ship_order(2000,10509,50);
```

Exercises in bb4.utc.edu

- Create three databases using the scripts from blackboard. File name is plsql.ch02.
- Start and test procedures or functions from p1.sql to p6.sql. File name is plsql.ch03.