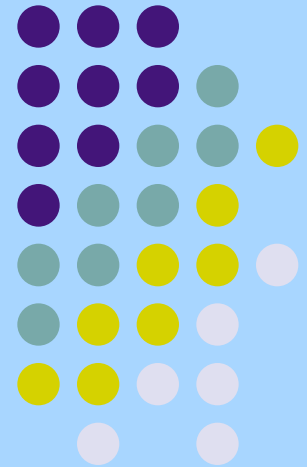



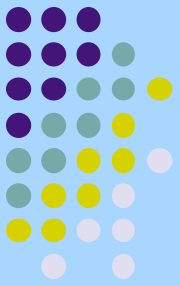
# Advanced SQL: Cursors & Stored Procedures

Instructor: Mohamed Eltabakh  
meltabakh@cs.wpi.edu

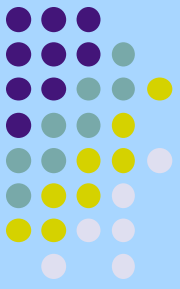


# Today's Roadmap

- Views
- Triggers
- Cursors 
- Stored Procedures



# Using Select Inside Triggers



```
Create Trigger EmpDate
Before Insert On Employee
For Each Row
```

```
Declare
temp date;
```

```
Begin
Select sysdate into temp from dual;
IF (:new.hireDate is null) Then
:new.hireDate := temp;
End IF;
End;
```

- Execute Select inside trigger
- Store the result in temp variables

Works fine if the Select returns one tuple...

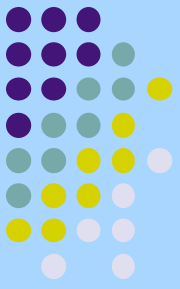
```
/
Create Trigger EmpDate
Before Insert On Employee
For Each Row
```

```
Declare
tempAge number;
tempName varchar2(100);
```

```
Begin
Select age, name into tempAge, tempName from R where ID = 5;
```

- Select two columns into two variables

```
End;
/
```



# Cursors: Introduction

- Select statement may return many records

```
Select emplID, name, salary  
From Employee  
Where salary > 120,000;
```



Get 0 or more records

- **What if inside a trigger:**
  - Want to execute a select statement
  - Get one record at a time
  - Do something with each record

**This's what a cursor  
does for you...**

# What is a Cursor



- A mechanism to navigate *tuple-by-tuple* over a relation
- Typically used inside triggers, stored procedures, or stored functions
- **Main Idea**
  - When we execute a query, a relation is returned
  - It is stored in private work area for the query
  - Cursor is a pointer to this area
  - Move the cursor to navigate over the tuples

# Creating a Cursor



Cursor name

Any query can go here

```
Cursor <name> IS <SQL query>;
```

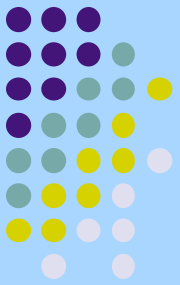
```
Cursor HighSalEmp IS
```

```
  Select empID, name, salary
```

```
  From Employee
```

```
  Where salary > 120,000;
```

# Cursor Operations



- **Create cursor**

**Cursor** *HighSalEmp IS*

*Select empID, name, salary  
From Employee  
Where salary > 120,000;*

- **Open cursor**

**Open** *HighSalEmp;*

- Execute the query and put the pointer at the first tuple

- **Fetch next tuple**

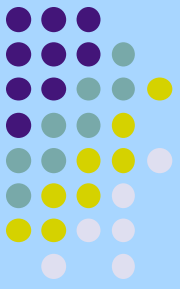
**Fetch** *HighSalEmp into <variable>;*

- Pointer moves automatically when a tuple is fetched

- **Close cursor**

**Close** *HighSalEmp;*

# Example 1



- Have two tables: Customer & Product
- *When insert a new customer*
  - *Put in Marketing table, the customer ID along with the products labeled 'OnSale'*

**Create Trigger** *NewCust*  
**After Insert On** *Customer*

**For Each Row**

**Declare**

pid number;

**cursor** C1 **is Select** product\_id **From** Product **Where** label = 'OnSale';

**Begin**

**open** C1;

**Loop**

**Fetch** C1 **Into** pid;

**IF** (C1%**Found**) **Then**

**Insert into** Marketing(Cust\_id, Product\_id) **values** (:new.Id, pid);

**END IF;**

**Exit When** C1%**NotFound**;

**END Loop;**

**close** C1;

**End;** /

Define the cursor in 'Declare' section

Open the cursor

Loop over each record at a time

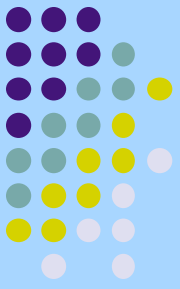
If the fetch returned a record

Customer ID

Close the cursor



# Example 2: Another way



- Use of the **FOR** loop with cursors

```
Create Trigger NewCust  
After Insert On Customer  
For Each Row  
Declare
```

```
  cursor C1 is Select product_id From Product Where label = 'OnSale';
```

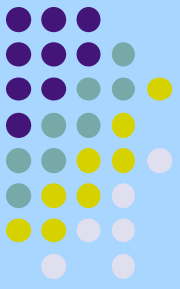
```
Begin  
  For rec In C1 Loop  
    Insert into Marketing(Cust_id, Product_id) values (:new.Id, rec.product_id);  
  End Loop;  
End; /
```

Automatically opens the cursor and fetches a record in each iteration

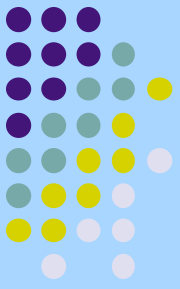
Automatically closes the cursor

That is how to read the rec variable

# Cursor Attributes



- These are attributes maintained by the system
- Assume **C1** is the cursor name
- **Attributes include:**
  - **C1%ROWCOUNT:** The number of tuples in C1
  - **C1%FOUND:** TRUE if the last fetch was successful
  - **C1%NOTFOUND:** TRUE if the last fetch was not successful
  - **C1%ISOPEN:** TRUE if C1 is open



# Parameterized Cursor

- Cursors can take parameters while opening them
- Very powerful to customize their execution each time
- *Example: Like the previous example, but select products with price < customer's budget*

```
Create Trigger NewCust  
After Insert On Customer
```

```
For Each Row
```

```
Declare
```

```
cursor C1 (budget number) is Select product_id From Product p  
Where p.label = 'OnSale' and p.price < budget;
```

Define the cursor with a parameter

```
Begin
```

```
For rec In C1(:new.budget) Loop
```

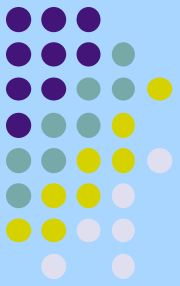
Pass the value at open time

```
Insert into Marketing(Cust_id, Product_id) values (:new.Id, rec.product_id);
```

```
End Loop;
```

```
End; /
```

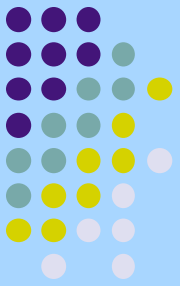
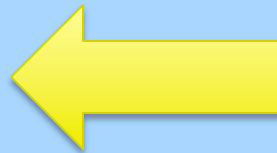
# Summary of Cursors



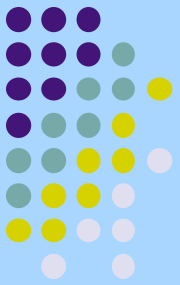
- **Efficient mechanism to iterate over a relation tuple-by-tuple**
- **Main operations**
  - Open, fetch, close
  - Usually used inside loops
- **Cursors can be parameterized**
  - What they return depends on the passed parameters

# Today's Roadmap

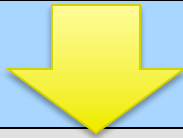
- Views
- Triggers
- Assertions
- Cursors
- Stored Procedures



# Stored Procedures & Functions

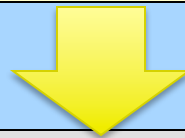


**Views**



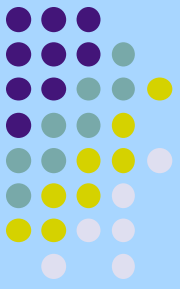
**Way to register queries inside DBMS**

**Stored Procedures &  
Functions**

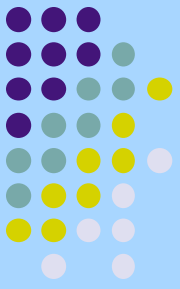


**Way to register code inside DBMS**

# Stored Procedures in Oracle



- Stored procedures in Oracle follow a language called **PL/SQL**
- PL/SQL: Procedural Language SQL
- Same language used inside DB triggers



# Creating A Stored Procedure

If exists, then drop it and create it again

'IS' or 'AS' both are valid

```
CREATE [OR REPLACE] PROCEDURE <procedureName> (<paramList>) [IS| AS]  
  <localDeclarations>  
Begin  
  <procedureBody>;  
End;  
/
```

A parameter in the paramList is specified as:

*<name> <mode> <type>*

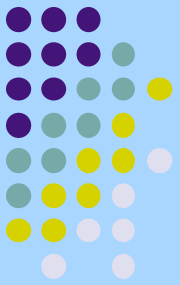
**Mode:**

**IN** → input parameter (default)

**OUT** → output parameter

**INOUT** → input and output parameter





# General Structure

```
CREATE [OR REPLACE] PROCEDURE procedure_name  
  [ (parameter [,parameter]) ]
```

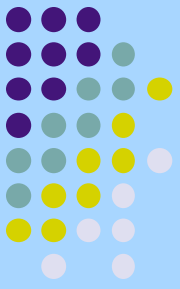
```
[IS | AS]  
  [declaration_section]
```

```
BEGIN  
  executable_section
```

Optional section for exception handling

```
[EXCEPTION ←  
  exception_section]
```

```
END [procedure_name];
```



# Example I

Define a variable

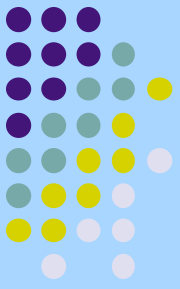
By default, it is IN

```
CREATE PROCEDURE remove_emp (employee_id NUMBER) AS
  tot_emps NUMBER;
BEGIN
  DELETE FROM employees
  WHERE employees.employee_id = remove_emp.employee_id;
  tot_emps := tot_emps - 1;
END;
```

You can use the procedure name before the parameter name

In PL/SQL a ';' ends a line without execution

Execute the command and create the procedure



# Example II

**Declaration section**

Define a cursor that references the input parameter

When anything goes wrong, it will come to Exception section

```
CREATE OR REPLACE Procedure UpdateCourse
```

```
( name_in IN varchar2 )
```

```
IS
```

```
  cnumber number;
```

```
  cursor c1 is
```

```
  select course_number
```

```
  from courses_tbl
```

```
  where course_name = name_in;
```

```
BEGIN
```

```
  open c1;
```

```
  fetch c1 into cnumber;
```

```
  if c1%notfound then
```

```
    cnumber := 9999;
```

```
  end if;
```

```
  insert into student_courses
```

```
  ( course_name,
```

```
    course_number )
```

```
  values ( name_in,
```

```
          cnumber );
```

```
  commit;
```

```
  close c1;
```

```
EXCEPTION
```

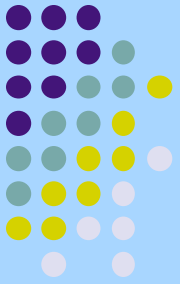
```
WHEN OTHERS THEN
```

```
  raise_application_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR-
```

```
'||SQLERRM);
```

```
END;
```

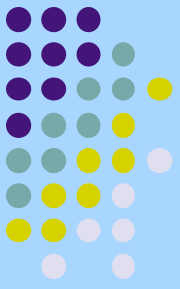
# Calling a Stored Procedure



- **SQL> exec <procedureName> [(*<paramList>*)];**

```
CREATE PROCEDURE remove_emp (employee_id NUMBER) AS
  tot_emps NUMBER;
BEGIN
  DELETE FROM employees
  WHERE employees.employee_id = remove_emp.employee_id;
  tot_emps := tot_emps - 1;
END;
/
```

```
SQL > exec remove_emp (10);
```



# Printing From Stored Procedures

Taking three parameters

```
SQL>
SQL> create or replace
2 procedure three_params(
3   p_p1 number,
4   p_p2 number,
5   p_p3 number ) as
6 begin
7   dbms_output.put_line( 'p_p1 = ' || p_p1 );
8   dbms_output.put_line( 'p_p2 = ' || p_p2 );
9   dbms_output.put_line( 'p_p3 = ' || p_p3 );
10 end three_params;
11 /
```

Procedure created.

Printing lines to  
output screen

**For the output to appear on screen. Make sure to run:**

Sql > set serveroutput on;

# Features in Stored Procedures



```
Create Procedure profiler_control(start_stop IN VARCHAR2,  
run_comm IN VARCHAR2,  
ret OUT number) AS  
  
ret_code INTEGER;  
  
BEGIN  
ret_code := 10;  
  
IF start_stop NOT IN ('START','STOP') THEN  
    ret:= 0;  
ELSIF start_stop = 'START' THEN  
    ret:= 1;  
ELSE  
    ret:= ret_code;  
END IF;  
END profiler_control;  
/
```

IN parameters

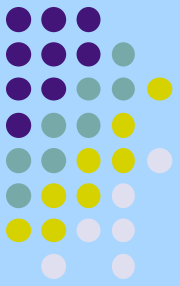
OUT parameters

Variable declaration

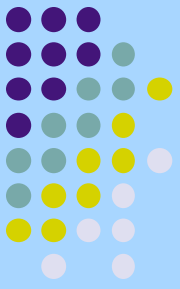
Variable assignment

IF statement

# More Features: CURSOR & FOR Statement



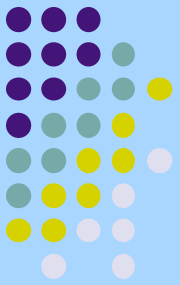
```
Create Procedure OpeningBal (p_type IN string) AS  
  cursor C1 Is  
    Select productId, name, price  
    From products  
    where type = p_type;  
Begin  
  For rec in C1 Loop  
    Insert into Temp values (rec.productId, rec.name, rec.price);  
  End Loop;  
End;  
/
```



# Return Value

- Stored procedures can set output variables
- Stored procedures do not return values
- Stored functions differ from procedure in that they return values





# Stored Functions

- Similar to stored procedures except that they return value

```
CREATE [OR REPLACE] FUNCTION <functionName>  
                                RETURN <type> [(<paramList>)] AS  
  
<localDeclarations>  
<functionBody>;
```

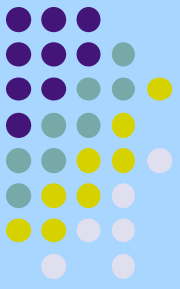
```
CREATE FUNCTION get_bal(acc_no IN NUMBER)  
  RETURN NUMBER  
  IS acc_bal NUMBER(11,2);  
  BEGIN  
    SELECT order_total  
    INTO acc_bal  
    FROM orders  
    WHERE customer_id = acc_no;  
    RETURN(acc_bal);  
  END;
```

The function return a number

Select into a variable

Return to the called

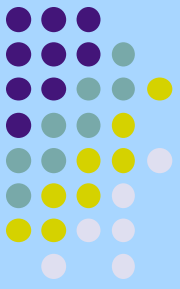
# Stored Functions



- All features in stored procedures are valid in stored functions
- Functions have an extra '*Return*' statement

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
  RETURN NUMBER
  IS acc_bal NUMBER(11,2);
BEGIN
  SELECT order_total
  INTO acc_bal
  FROM orders
  WHERE customer_id = acc_no;
  RETURN(acc_bal);
END;
```

# Using Stored Procedures or Functions



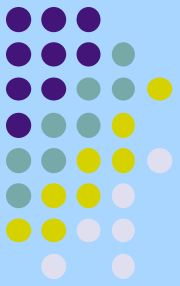
- **Stored Procedures**

- Called from other procedures, functions, triggers, or standalone

- **Stored Functions**

- In addition to above, can be used inside **SELECT** statement
  - In **WHERE**, **HAVING**, or **projection** list

# Example I



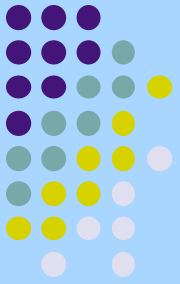
```
CREATE FUNCTION MaxNum() RETURN number AS  
    num1 number;  
BEGIN  
    SELECT MAX (sNumber) INTO num1 FROM Student;  
    RETURN num1;  
END;  
/
```

```
SQL> Select * from Student where sNumber = MaxNum();
```

Calling the function in the Where clause  
(function will be executed once)

# Example II

Adding a parameter

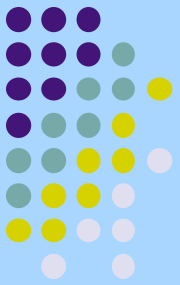


```
CREATE FUNCTION MaxNum(lastName_in varchar2)
RETURN number AS
    num1 number;
BEGIN
    SELECT MAX (sNumber) INTO num1 FROM Student
    Where lastName = lastName_in;
    RETURN num1;
END;
/
```

```
SQL> Select * from Student S where S.sNumber = MaxNum(S.lastName);
```

Calling the function in the Where clause  
(function will execute with each record in Student)

# Example III



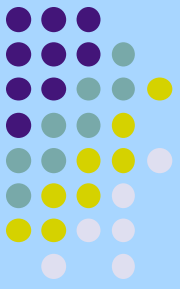
```
CREATE FUNCTION MaxNum(lastName_in varchar2)
RETURN number AS
    num1 number;
BEGIN
    SELECT MAX (sNumber) INTO num1 FROM Student
    Where lastName = lastName_in;
    RETURN num1;
END;
/
```

```
SQL> Select MaxNum(S.lastName) from Student S;
```



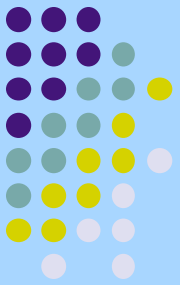
Calling the function in the projection list

# Summary of Stored Procedures/ Functions



- Code modules that are stored inside the DBMS
- Used and called repeatedly
- Powerful programming language style
- Can be called from other procedures, functions, triggers, or from select statement (only functions)

# End of Advanced SQL



- **Views**
- **Triggers**
- **Assertions**
- **Cursors**
- **Stored Procedures/Functions**

