SQL/PSM

Procedures Stored in the Database

General-Purpose Programming

1

Stored Procedures

- An extension to SQL, called SQL/PSM, or "persistent, stored modules," allows us to store procedures as database schema elements.
- The programming style is a mixture of conventional statements (if, while, etc.) and SQL.
- Let's us do things we cannot do in SQL alone.

2

Basic PSM Form

```
CREATE PROCEDURE <name> (
        <parameter list> )
   <optional local declarations>
   <body>;
```

- Function alternative:

```
CREATE FUNCTION <name> (
        <parameter list> ) RETURNS <type>
```

3

Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the *mode* can be:
  - IN = procedure uses value, does not change value.
  - OUT = procedure changes, does not use.
  - INOUT = both.

4

Example: Stored Procedure

- Let's write a procedure that takes two arguments *b* and *p*, and adds a tuple to Sells that has bar = 'Joe''s Bar', beer = *b*, and price = *p*.
  - Used by Joe to add to his menu more easily.

5

The Procedure

```
CREATE PROCEDURE JoeMenu (
   IN b      CHAR(20),
   IN p      REAL
)
INSERT INTO Sells
VALUES('Joe''s Bar', b, p);
```

Parameters are both read-only, not changed

The body --- a single insertion

6

Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments.
- Example:

```
CALL JoeMenu('Moosedrool', 5.00);
```

- Functions used in SQL expressions where a value of their return type is appropriate.

Types of PSM statements -- 1

- RETURN <expression> sets the return value of a function.
  - Unlike C, etc., RETURN *does not* terminate function execution.
- DECLARE <name> <type> used to declare local variables.
- BEGIN . . . END for groups of statements.
  - Separate by semicolons.

Types of PSM Statements -- 2

- Assignment statements:

```
SET <variable> = <expression>;
```

  - Example: SET b = 'Bud';
- Statement labels: give a statement a label by prefixing a name and a colon.

IF statements

- Simplest form:

```
IF <condition> THEN
    <statements(s)>
END IF;
```

- Add ELSE <statement(s)> if desired, as

```
IF . . . THEN . . . ELSE . . . END IF;
```

- Add additional cases by ELSEIF <statements(s)>:

```
IF … THEN … ELSEIF … ELSEIF … ELSE … END IF;
```

Example: IF

- Let's rate bars by how many customers they have, based on Frequents(drinker, bar).
  - <100 customers: 'unpopular'.
  - 100-199 customers: 'average'.
  - >= 200 customers: 'popular'.
- Function Rate(b) rates bar b.

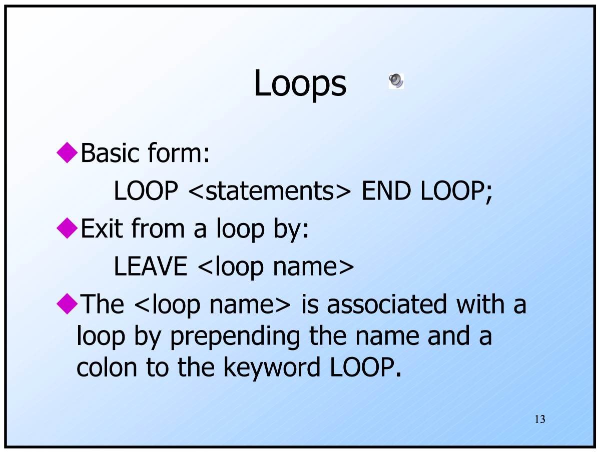
Example: IF (continued)

```
CREATE FUNCTION Rate (IN b CHAR(20) )
    RETURNS CHAR(10)
    DECLARE cust INTEGER;
  BEGIN
    SET cust = (SELECT COUNT(*) FROM Frequents
                  WHERE bar = b);
    IF cust < 100 THEN RETURN 'unpopular'
    ELSEIF cust < 200 THEN RETURN 'average'
    ELSE RETURN 'popular'
    END IF;
  END;
```

Number of customers of bar b

Nested IF statement

Return occurs here, not at one of the RETURN statements

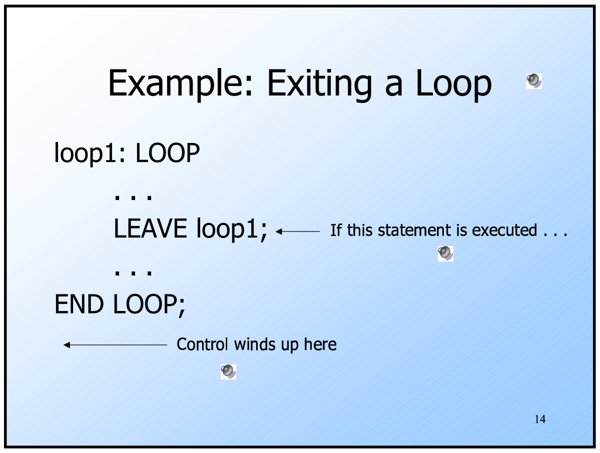


Loops

- Basic form:

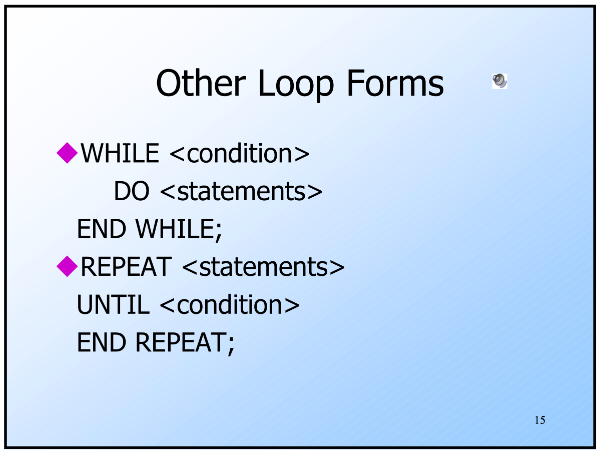
  LOOP <statements> END LOOP;
- Exit from a loop by:

  LEAVE <loop name>
- The <loop name> is associated with a loop by prepending the name and a colon to the keyword LOOP.



Example: Exiting a Loop

```
loop1: LOOP
      . . .
      LEAVE loop1;  ←  If this statement is executed . . .
      . . .
END LOOP;
←           Control winds up here
```



Other Loop Forms

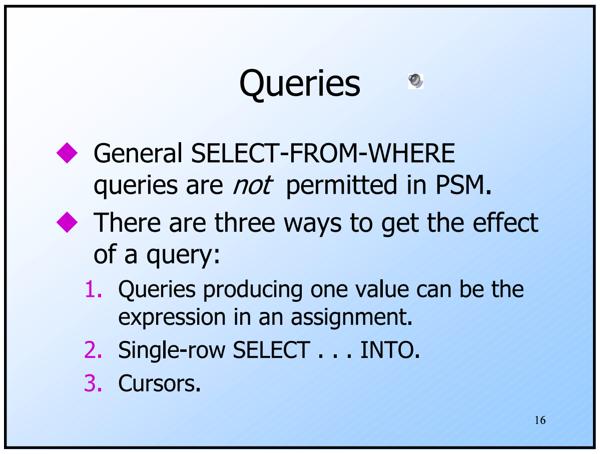
- WHILE <condition>

  DO <statements>

  END WHILE;
- REPEAT <statements>

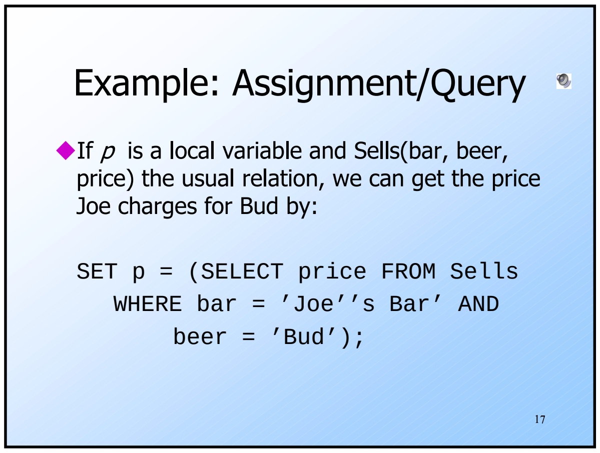
  UNTIL <condition>

  END REPEAT;



Queries

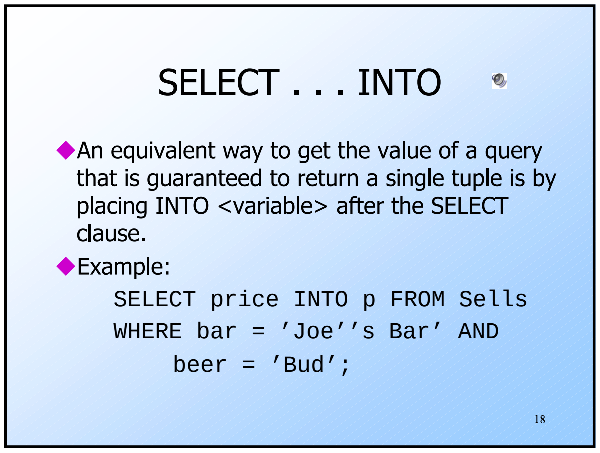
- General SELECT-FROM-WHERE queries are *not* permitted in PSM.
- There are three ways to get the effect of a query:
  1. Queries producing one value can be the expression in an assignment.
  2. Single-row SELECT . . . INTO.
  3. Cursors.



Example: Assignment/Query

- If *p* is a local variable and Sells(bar, beer, price) the usual relation, we can get the price Joe charges for Bud by:

```
SET p = (SELECT price FROM Sells
   WHERE bar = 'Joe''s Bar' AND
        beer = 'Bud');
```



SELECT . . . INTO

- An equivalent way to get the value of a query that is guaranteed to return a single tuple is by placing INTO <variable> after the SELECT clause.
- Example:

```
SELECT price INTO p FROM Sells
WHERE bar = 'Joe''s Bar' AND
     beer = 'Bud';
```

Cursors

- A *cursor* is essentially a tuple-variable that ranges over all tuples in the result of some query.
- Declare a cursor *c* by:

```
DECLARE c CURSOR FOR <query>;
```

19

Opening and Closing Cursors

- To use cursor *c*, we must issue the command:

```
    OPEN c;
```

  - The query of *c* is evaluated, and *c* is set to point to the first tuple of the result.
- When finished with *c*, issue command:

```
    CLOSE c;
```

20

Fetching Tuples From a Cursor

- To get the next tuple from cursor c, issue command:

```
    FETCH c INTO x1, x2,…,xn ;
```

- The *x*'s are a list of variables, one for each component of the tuples referred to by *c*.
- c is automatically moved to the next tuple.

21

Breaking Cursor Loops -- 1

- The usual way to use a cursor is to create a loop with a FETCH statement, and do something with each tuple fetched.
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver.

22

Breaking Cursor Loops -- 2

- Each SQL operation returns a *status*, which is a 5-digit number.
  - For example, 00000 = "Everything OK," and 02000 = "Failed to find a tuple."
- In PSM, we can get the value of the status in a variable called SQLSTATE.

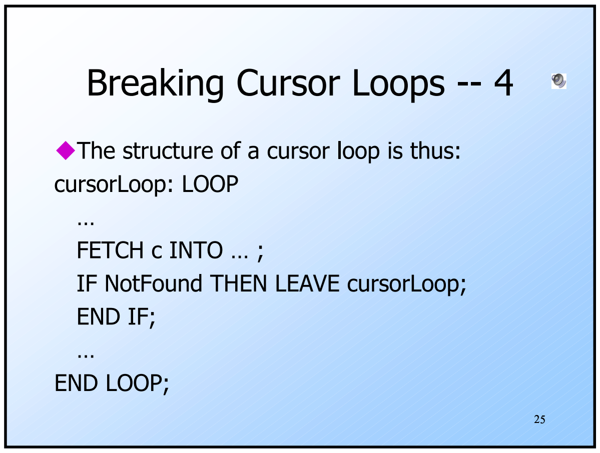
23

Breaking Cursor Loops -- 3

- We may declare a condition, which is a boolean variable that is true if and only if SQLSTATE has a particular value.
- Example: We can declare condition NotFound to represent 02000 by:

```
DECLARE NotFound CONDITION FOR
  SQLSTATE '02000';
```

24

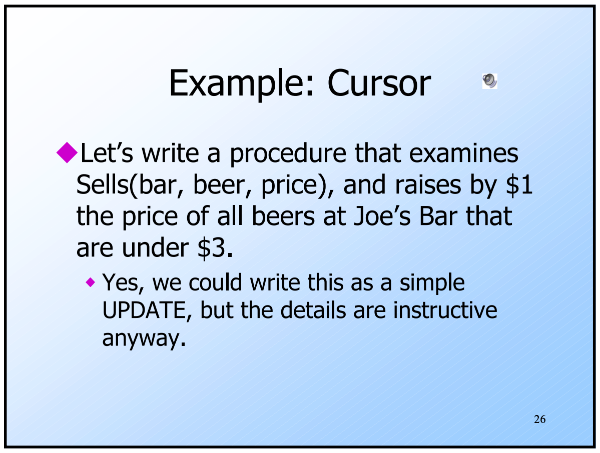


Breaking Cursor Loops -- 4

- The structure of a cursor loop is thus:

```
cursorLoop: LOOP
  …
  FETCH c INTO … ;
  IF NotFound THEN LEAVE cursorLoop;
  END IF;
  …
END LOOP;
```

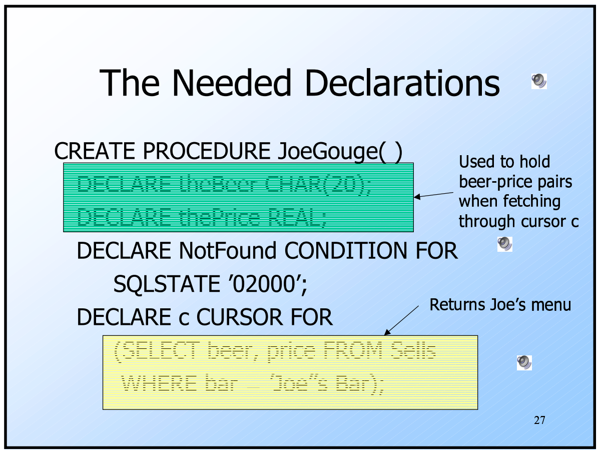
25



Example: Cursor

- Let's write a procedure that examines Sells(bar, beer, price), and raises by $1 the price of all beers at Joe's Bar that are under $3.
  - Yes, we could write this as a simple UPDATE, but the details are instructive anyway.

26

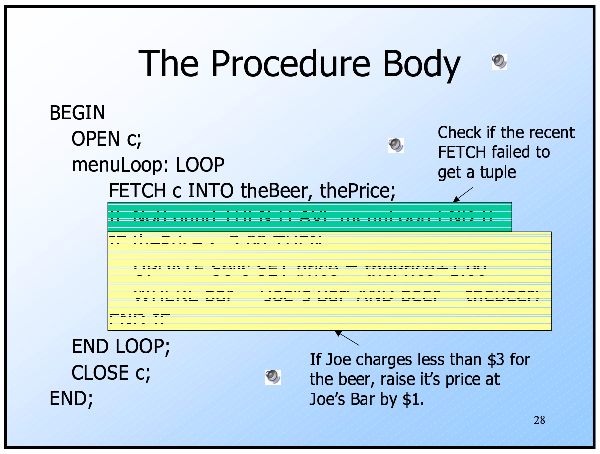


The Needed Declarations

```
CREATE PROCEDURE JoeGouge( )
  DECLARE theBeer CHAR(20);
  DECLARE thePrice REAL;
  DECLARE NotFound CONDITION FOR
     SQLSTATE '02000';
  DECLARE c CURSOR FOR
    (SELECT beer, price FROM Sells
     WHERE bar = 'Joe''s Bar);
```

Used to hold beer-price pairs when fetching through cursor c
Returns Joe's menu

27



The Procedure Body

```
BEGIN
  OPEN c;
  menuLoop: LOOP
     FETCH c INTO theBeer, thePrice;
     IF NotFound THEN LEAVE menuLoop END IF;
     IF thePrice < 3.00 THEN
       UPDATE Sells SET price = thePrice+1.00
       WHERE bar = 'Joe''s Bar' AND beer = theBeer;
     END IF;
  END LOOP;
  CLOSE c;
END;
```

Check if the recent FETCH failed to get a tuple
If Joe charges less than $3 for the beer, raise it's price at Joe's Bar by $1.

28