

Parameters

Parameter declaration:

MODE name type

Mode:

- IN – value comes in; any changes in the body of the procedure has no effect on the parameter's value
- OUT – procedure changes value and passes back to caller
- INOUT – value is passed in and changed value is passed back to caller

Notes:

- name cannot be the same as a table; must qualify column names if a parameter has the same name
- type should be compatible with actual column type, but doesn't have to be exactly the same

Example – Parameters

Retrieve all reservations (and related information) for a particular sailor (by name) from the sailors database –

```
CREATE PROCEDURE ReservationsFor
  ( IN sailor VARCHAR(30) )
BEGIN
  SELECT *
  FROM SAILOR NATURAL JOIN RESERVATION
  NATURAL JOIN BOAT
  WHERE Sname=sailor;
END
```

- call
CALL ReservationsFor('Dustin')

Example – Parameters

Delete a sailor and that sailor's reservations –

```
CREATE PROCEDURE DeleteSailor ( IN delsid INT )
BEGIN
  DELETE FROM RESERVATION WHERE Sid=delsid;
  DELETE FROM SAILOR WHERE Sid=delsid;
END
```

Example – OUT Parameters

Retrieve the number of reservations for a particular sailor from the sailors database –

```
CREATE PROCEDURE NumReservations( IN sailor VARCHAR(30),
                                  OUT numres INT )
BEGIN
  SELECT COUNT(*)
  FROM SAILOR NATURAL JOIN RESERVATION NATURAL JOIN BOAT
  WHERE Sname=sailor
  INTO numres;
END
```

(better done with a stored function)

Accessing OUT Parameters

Call:

```
CALL name ( vars )  
  - @varname to reference a variable
```

Access result:

```
SELECT @varname
```

Example – OUT Parameters

Call the stored procedure and access the result:

```
CALL NumReservations('Dustin',@num);  
SELECT @num;  
  - the number of reservations ends up in @num
```

@num can be used in places a value is expected:

```
CALL NumReservations('Dustin',@num);  
SELECT Sname,COUNT(*) AS 'Number of Reservations'  
FROM RESERVATION NATURAL JOIN SAILOR  
GROUP BY Sid  
HAVING COUNT(*) = @num;  
  - retrieve names and number of reservations for each sailor  
  having the same number of reservations as Dustin
```

Stored Functions

- define

```
CREATE FUNCTION name ( parameters )  
  RETURNS type  
  BEGIN  
    statements  
  RETURN value;  
END
```

– parameters can only be IN, so no mode is included in the declaration

- call

```
SELECT name(values)  
  - function can also be called in other places where a value is  
  appropriate
```

Variables

Declaration:

```
DECLARE name type [DEFAULT value];  
  - goes at the beginning of the stored routine
```

Set value:

```
SET name = expr;
```

Set value from a query:

```
SELECT columns  
FROM ...  
INTO vars  
  - number and order of vars must match columns  
  - can only have a single row produced
```

Example – Functions

Function version of the NumReservations procedure:

```
CREATE FUNCTION NumReservations2 ( sailor VARCHAR(30) )
RETURNS INT
BEGIN
    DECLARE numres INT DEFAULT -1;
    SELECT COUNT(*)
    FROM SAILOR NATURAL JOIN RESERVATION NATURAL JOIN BOAT
    WHERE Sname=sailor
    INTO numres;
    RETURN numres;
END
```

Example – Functions

Call the stored function and access the result:

```
SELECT NumReservations2('Dustin');
```

The function can also be used in other places a value is expected:

```
SELECT Sname,COUNT(*) AS 'Number of Reservations'
FROM RESERVATION NATURAL JOIN SAILOR
GROUP BY Sid
HAVING COUNT(*) = NumReservations2('Dustin');
```

- retrieve names and number of reservations for each sailor having the same number of reservations as Dustin

Conditionals

```
IF expression THEN commands
ELSEIF expression THEN commands
ELSE commands
END IF;
```

```
CASE case_expression
    WHEN when_expression THEN commands
    WHEN when_expression THEN commands
    .
    .
    .
    ELSE commands
END CASE;
```

Loops

```
WHILE expression DO
    statements
END WHILE
```

```
REPEAT
    statements
UNTIL expression
END REPEAT
```

There is also a more general LOOP syntax, with ways to specify when to exit the loop and when to do another iteration.

Cursors

Cursors allow you to iterate through results returned by a SELECT query.

- declare

```
DECLARE name CURSOR FOR SELECT ...
```

– must be after variable declarations

```
DECLARE CONTINUE HANDLER FOR NOT FOUND ...
```

- open

```
OPEN name;
```

- fetch the next row

```
FETCH name INTO var;
```

- close

```
CLOSE name;
```

Example

count the number of reservations in the database

(to illustrate loops and cursors – a simpler solution is just SELECT COUNT(*) FROM RESERVATION)

```
CREATE FUNCTION CountReservations() RETURNS INT
BEGIN
  DECLARE done BOOLEAN DEFAULT FALSE;
  DECLARE cursid INT;
  DECLARE count INT DEFAULT 0;
  DECLARE curres CURSOR FOR
    SELECT Sid FROM RESERVATION;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET done = TRUE;

  OPEN curres;
  FETCH curres INTO cursid;
  WHILE NOT done DO
    SET count = count+1;
    FETCH curres INTO cursid;
  END WHILE;
  CLOSE curres;
  RETURN count;
END
```

Example

```
CREATE FUNCTION CountReservations() RETURNS INT
BEGIN
```

```
  DECLARE done BOOLEAN DEFA
  DECLARE cursid INT;
  DECLARE count INT DEFAULT
  DECLARE curres CURSOR FOR
    SELECT Sid FROM RESERVA
  DECLARE CONTINUE HANDLER
    SET done = TRUE;
```

```
  OPEN curres;
```

```
  FETCH curres INTO cursid;
  WHILE NOT done DO
    SET count = count+1;
    FETCH curres INTO cursid;
  END WHILE;
```

```
  CLOSE curres;
```

```
  RETURN count;
```

```
END
```

the sequencing is vital here – when FETCH triggers the NOT FOUND condition, done is set to TRUE but execution then continues normally – which mean the loop does not exit until the next time the condition is checked

the following loop would count one too many, because the SET is still executed after the FETCH regardless of the outcome of the FETCH – done isn't checked until the loop cycles around

```
  WHILE NOT done DO
    FETCH curres INTO cursid;
    SET count = count+1;
  END WHILE;
```

Handlers

Declare handler:

```
DECLARE action HANDLER FOR condition
  statement
```

– must be after variable declarations

'Action' specifies what is done after the handler statement is executed.

- CONTINUE – continue execution of the program
- EXIT – terminate execution of the BEGIN ... END block where the handler is declared

'Condition' can be

- a MySQL error code (a number)
- an SQLSTATE value (5-character string literal)
- name of a previously-declared condition

Handler Conditions

Shortcuts:

- SQLWARNING – SQLSTATE values beginning with 01
- NOT FOUND – SQLSTATE values beginning with 02
 - 02000 means 'no data'
- SQLEXCEPTION – SQLSTATE values not beginning with 00, 01, or 02

See MySQL documentation for a full list.

- <https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html>

Example

delete all sailors with the specified name
(and their reservations)

```
CREATE PROCEDURE DeleteAllSailorsByName
( IN delname VARCHAR(45) )
BEGIN
  DECLARE delsid INT;
  DECLARE done BOOLEAN DEFAULT FALSE;

  DECLARE sailors CURSOR FOR
    SELECT Sid FROM SAILOR WHERE Sname=delname;
  DECLARE CONTINUE HANDLER FOR NOT FOUND ← 'NOT FOUND'
    SET done = TRUE;                                     condition is
                                                         triggered by
                                                         FETCH when there
                                                         are no more rows

  OPEN sailors;
  FETCH sailors INTO delsid;
  WHILE NOT done DO
    DELETE FROM RESERVATION WHERE Sid=delsid;
    DELETE FROM SAILOR WHERE Sid=delsid;
    FETCH sailors INTO delsid;
  END WHILE;
  CLOSE sailors;
END
```

Example

delete a sailor by name, doing nothing if
there is more than one sailor with the name

```
CREATE PROCEDURE DeleteSailorByName
( IN delname VARCHAR(45) )
BEGIN
  DECLARE delsid INT;
  DECLARE EXIT HANDLER FOR SQLSTATE '42000'
    BEGIN SELECT 'too many sailors!'; END; ← end with error
                                                         message if
                                                         SQLSTATE
                                                         42000 occurs

  SELECT Sid
  FROM SAILOR
  WHERE Sname=delname
  INTO delsid; ← SELECT ... INTO
                                                         triggers a 42000
                                                         SQLSTATE if
                                                         more than one
                                                         row is found

  DELETE FROM RESERVATION WHERE Sid=delsid;
  DELETE FROM SAILOR WHERE Sid=delsid;
END
```

Permissions for Stored Objects

- the *security context* determines whether execution is with the privileges of the DEFINER account (regardless of the invoker) or only with the privileges of the invoker
 - both views and stored routines support the SQL SECURITY characteristic to set the security context
 - value is DEFINER or INVOKER (defaults to DEFINER)
 - triggers always run as DEFINER

```
CREATE DEFINER = 'admin'@'localhost' PROCEDURE p1()
SQL SECURITY DEFINER
BEGIN
  UPDATE t1 SET counter = counter + 1;
END;
```

```
CREATE DEFINER = 'admin'@'localhost' PROCEDURE p2()
SQL SECURITY INVOKER
BEGIN
  UPDATE t1 SET counter = counter + 1;
END;
```

- the DEFINER account defaults to the creator
 - specifying a different DEFINER account requires the SET_USER_ID privilege