CIS 363 MySQL

Chapter 17 Prepared Statements

Chapter 18 Store Prodcures and Functions

Ch 17. Prepared Statements

- MySQL Server supports prepared statements, which are useful when you want to run several queries that differ only in very small details.
- Benefits of Prepared Statements
  - Convenience and Efficiency
  - Enhanced performance: the complete statement is parsed only once by the server.
  - Less traffic between the server and the client: When the parse is complete, the server and client may make use of a new protocol that requires fewer data conversions.

Ch 17. Prepared Statements

- In most circumstances, statements are prepared and executed using the programing interface that you normally use for writing applications that use MySQL. However, to aid in testing and debugging, it is possible to define and use prepared statements from within the mysql command-line client.

- Example:

```
mysql> PREPARE my_stmt FROM
    'SELECT COUNT(*) FROM CountryLanguage WHERE CountryCode = ?';
```

-- Compiled & Optimized Once.

--NOTE: Prepares/pre-compiles above statement and stores the compiled query object in my_stmt. Each '?' is a place holder for variable data that will be provided later.

Ch 17. Prepared Statements

```
mysql> SET @code = 'ESP';
mysql> EXECUTE my_stmt USING @code;

+----------------+
| COUNT(*)   |
+----------------+
|       4          |
+----------------+
1 row in set (0.00 sec)
```

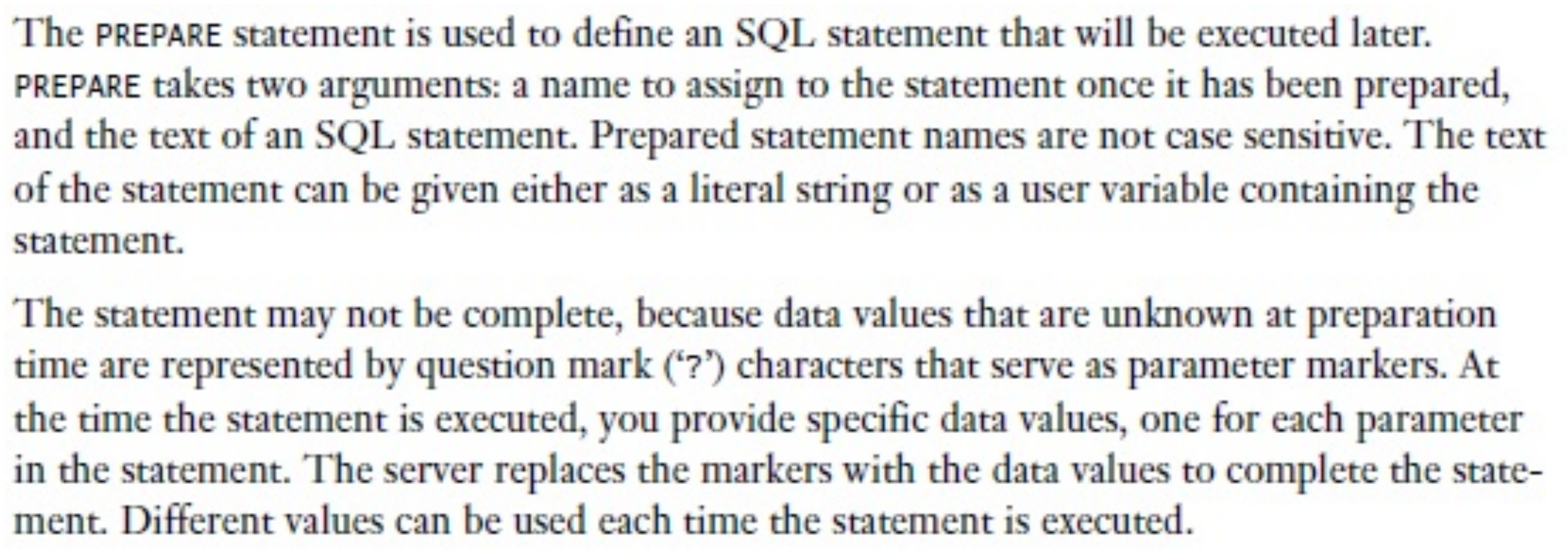
Versus: SELECT COUNT(*) FROM CountryLanguage WHERE CountryCode = 'ESP' -- Compiled & Optimized Once.

```
mysql> SET @code = 'RUS';
mysql> EXECUTE my_stmt USING @code;  -- Just execute, not re-compile or re-optimized.
+---------------+
| COUNT(*)  |
+---------------+
|      12        |
+---------------+
1 row in set (0.00 sec)
```

Versus: SELECT COUNT(*) FROM CountryLanguage WHERE CountryCode = 'RUS' -- Compiled/Optimized a second time.

```
mysql> DEALLOCATE PREPARE my_stmt;
```

Ch 17. Prepared Statements



The PREPARE statement is used to define an SQL statement that will be executed later. PREPARE takes two arguments: a name to assign to the statement once it has been prepared, and the text of an SQL statement. Prepared statement names are not case sensitive. The text of the statement can be given either as a literal string or as a user variable containing the statement.

The statement may not be complete, because data values that are unknown at preparation time are represented by question mark ('?') characters that serve as parameter markers. At the time the statement is executed, you provide specific data values, one for each parameter in the statement. The server replaces the markers with the data values to complete the statement. Different values can be used each time the statement is executed.

Ch 17. Prepared Statements

```
mysql> PREPARE namepop FROM  'SELECT Name, Population FROM Country WHERE Code = ?';
Query OK, 0 rows affected (0.09 sec)
Statement prepared   -- & Optimized Just ONCE!!!

mysql> PREPARE error FROM 'SELECT NonExistingColumn FROM Country WHERE Code = ?';
ERROR 1054 (42S22): Unknown column 'NonExistingColumn' in 'field list‘
```

- If you PREPARE a statement using a statement name that already exists, the server fisrt discards the prepared statement currently associated with the name, and the prepares the new statement. If the new statement contains an error and can not be prepared, the result is that no statement with the given name will exist.

For example:
Q: After you execute the following statements, how many prepared statements exist?

```
PREPARE s1 FROM 'SELECT 1';
PREPARE s2 FROM 'SELECT 2';
PREPARE s1 FROM 'SELECT (1+2';
```

A: One prepared statement exists. After the first two PREPARE statements, two prepared statements exist (s1 and s2). The third statement causes the original s1 to be discarded because it uses the same statement name. However, it does not result in a new prepared statement because the statement contains a syntax error. Only s2 exists after all three PREPARE statements have been executed.

Ch 17. Prepared Statements

- After a statement has been prepared, it can be executed. If the statement contains any ‘?’ parameter markers, a data value must be supplied for each of them by means of user variables.

```
mysql> SET @var1 = 'USA';
mysql> EXECUTE namepop USING @var1;  -- Just executed, not re-compiled or re-optimized
+-------------------+--------------------+
| Name              | Population       |
+-------------------+--------------------+
| United States  |  278357000     |
+-------------------+--------------------+

mysql> SET @var2 = 'CAN';
mysql> EXECUTE namepop USING @var2; -- Just executed, not re-compiled or re-optimized
+------------+----------------+
| Name     | Population   |
+------------+----------------+
| Canada  |   31147000  |
+------------+----------------+

mysql> SET @var3 = 'GBR';
mysql> EXECUTE namepop USING @var3; -- Just executed, not re-compiled or re-optimized
+----------------------+-----------------+
| Name                   | Population   |
+----------------------+-----------------+
| United Kingdom  |   59623400  |
+----------------------+-----------------+
```

Ch 17. Prepared Statements

- If you refer to a use variable that has not been initialized, its value is *NULL*.

```
mysql> EXECUTE name pop USING @var4;
Empty set (0.00sec)
```

- MySQL does not allow every type of SQL statement to be prepared. Those that may be prepared are limited to the following:
  - SELECT statement
  - Statements that modify data; INSERT, REPLACE, UPDATE, and DELETE
  - CREATE TABLE statement
  - SET, DO, and many SHOW statements

- A prepared statement exist only for the duration of the session in which it is created, and it is visible only to the session in which it is created. When a session ends, all prepared statements for that session are discarded. Thus, there is rarely any reason to drop prepared statements explicitly. If you want to do so, use the DEALLOCATE PREPARE statement . MySQL also provides DROP PREPARE statement. ex: mysql> DEALLOCATE PREPARE namepop;

Ch 18. Stored Procedures and Functions

STORED PROCEDURES & FUNCTIONS (Routines):

- MySQL code that is stored directly in the database instead of on a client application or file. Stored procedures/functions (routines) have several advantages over sql files and client embedded SQL. First, the processing of commands may be done in the DBMS and therefore on the server. In client-server applications, processing is often shared between the Client and Server, which leads to Network traffic. Secondly stored procedures/ functions are available to all leading to consistency of business rules and less re-invention of the wheel. They may also reduce development time by reducing the amount of code that must be written into applications. This also allows database rules to be enforced where they should be enforced, on the database. Trusting the client to enforce the integrity of the database is a little like trusting the fox with the key to the hen house.

Ch 18. Stored Procedures and Functions

Benefits of Stored Routines: p.282&283

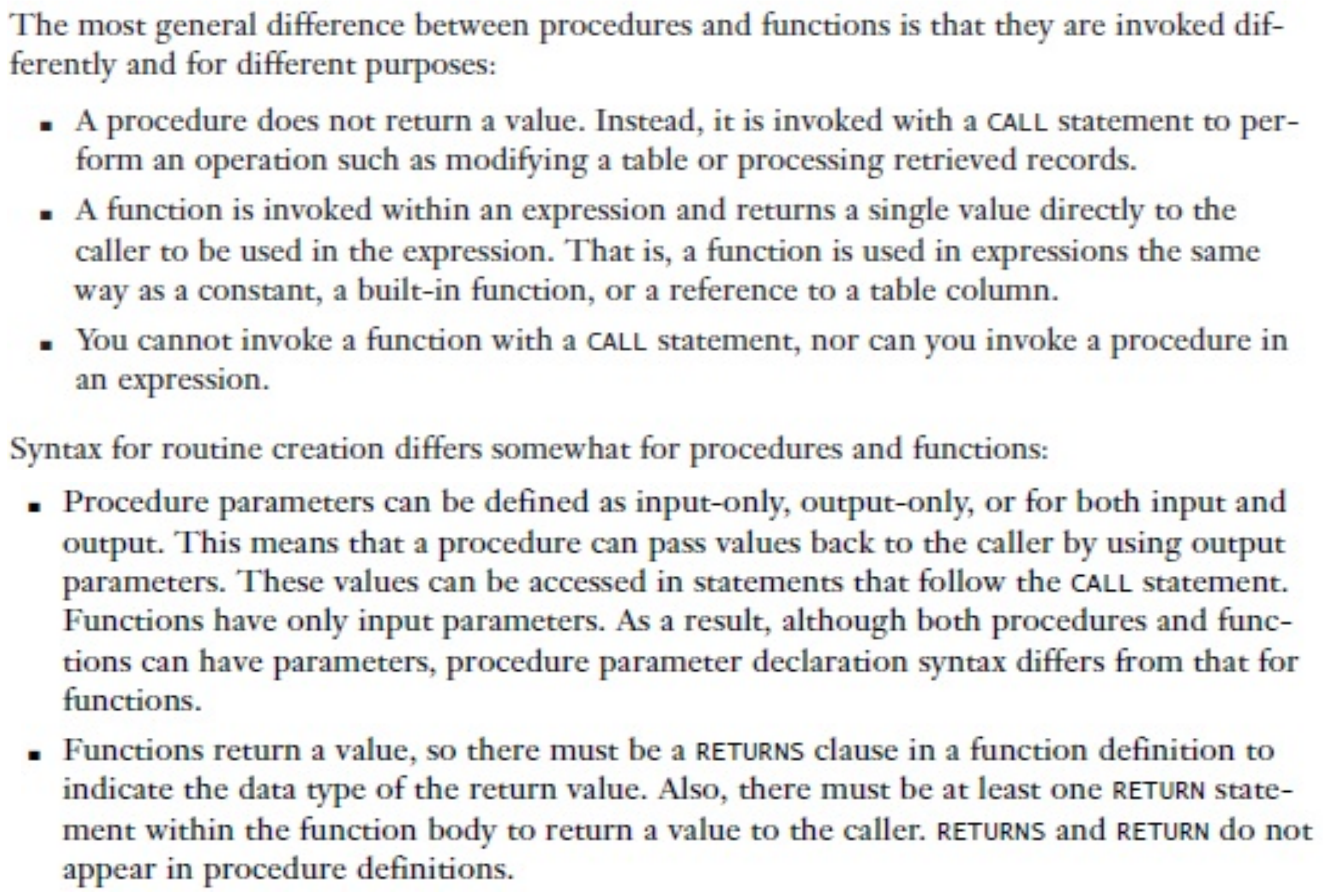
- More Flexible SQL Syntax: Store routines can be written using extensions to SQL syntax such as compound statements and flow-control constructs, that make it easier to express complex logic.
- Error handling capabilities: A stored routines can create error handlers to be used when exceptional conditions arise. The occurrence of an error need not cause termination of the routine but can be handled appropriately.
- Standard compliance. The MySQL implementation of stored routines conforms to standard SQL syntax. (however, there are some exceptions such as Cursors)
- Code packaging and encapsulation: A routine allows the code that performs an operation to be stored once on the server and accessed from multiple applications. The code need not be included within multiple applications.
- Less “re-invention of the wheel”: A collection of stored acts as a library of solutions to problems. Developers can use them rather than re-implement the code from scratch.
- Separation of logic: Factoring out of the logic of specific application operations into stored routines reduces the complexity of an application’s own logic and makes it easier to understand. (continue)

Ch 18. Stored Procedures and Functions

- Easy of maintenance: A single copy of a routine is easier to maintain than a copy embedded within each application. Upgrading applications is easier if they all use a routine in common.
- Reduction in network bandwidth requirement: If the operation is performed within a stored routine, intermediate statements and results are processed entirely on the server side and do not cross the network. This improves performance and results in less contention for resources, particularly for busy for low bandwidth networks.
- Server upgrades benefit clients. Upgrades to the server host improve the performance of stored routines that executes on that host. This improves performance for client applications that use the routines even though the client machines are not upgraded.
- Better security: A routine can be written to access sensitive data on the definer's behalf for the invoker, but not return anything that the invoker should not see. A routine can also be used to modify tables in a safe way, without giving users direct to the tables. This prevents them from making possibly unsafe changes themselves.

Ch 18. Stored Procedures and Functions

Differences Between Stored Procedures and Functions.



The most general difference between procedures and functions is that they are invoked differently and for different purposes:

- A procedure does not return a value. Instead, it is invoked with a `CALL` statement to perform an operation such as modifying a table or processing retrieved records.
- A function is invoked within an expression and returns a single value directly to the caller to be used in the expression. That is, a function is used in expressions the same way as a constant, a built-in function, or a reference to a table column.
- You cannot invoke a function with a `CALL` statement, nor can you invoke a procedure in an expression.

Syntax for routine creation differs somewhat for procedures and functions:

- Procedure parameters can be defined as input-only, output-only, or for both input and output. This means that a procedure can pass values back to the caller by using output parameters. These values can be accessed in statements that follow the `CALL` statement. Functions have only input parameters. As a result, although both procedures and functions can have parameters, procedure parameter declaration syntax differs from that for functions.
- Functions return a value, so there must be a `RETURNS` clause in a function definition to indicate the data type of the return value. Also, there must be at least one `RETURN` statement within the function body to return a value to the caller. `RETURNS` and `RETURN` do not appear in procedure definitions.

Ch 18. Stored Procedures and Functions

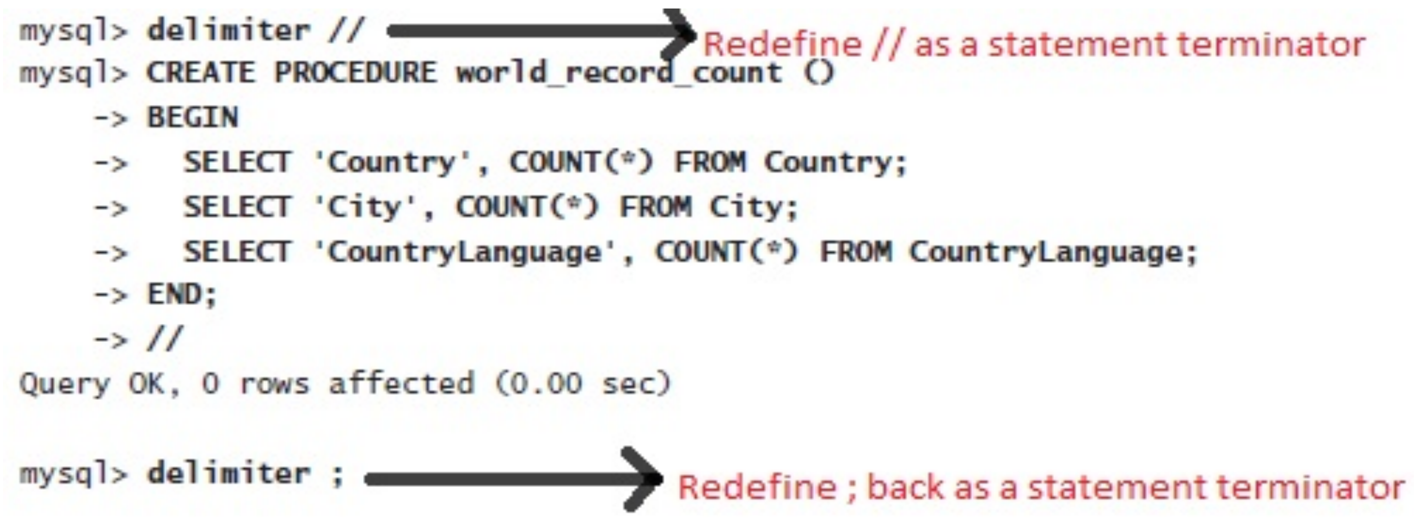
The Namespace for Stored Routines

- Each stored routine is associated with a particular database, just like a table or a view.
- MySQL interprets an unqualified reference, *routine_name*, as a reference to a procedure or function in the default database. To refer to a routine in a specific database, use a qualified name of the form *db_name.routine_name*.
- When routines executes, it implicitly changes the defualt database to the database associated with the routine and restores the previous default database when it terminates. Due to this association of a routine with a database, you must have access to that database to be able to invoke the routine.
- When you drop a database, any stored functions in the database are also dropped.
- Stored procedures and functions do not share the same namespace. It is not possible to have two procedures or two functions with the same name in a database, but it is possible to have a procedure and a function with the same name in a database. (not recommended because it is very confusing)

Ch 18. Stored Procedures and Functions

Defining Stored Routines

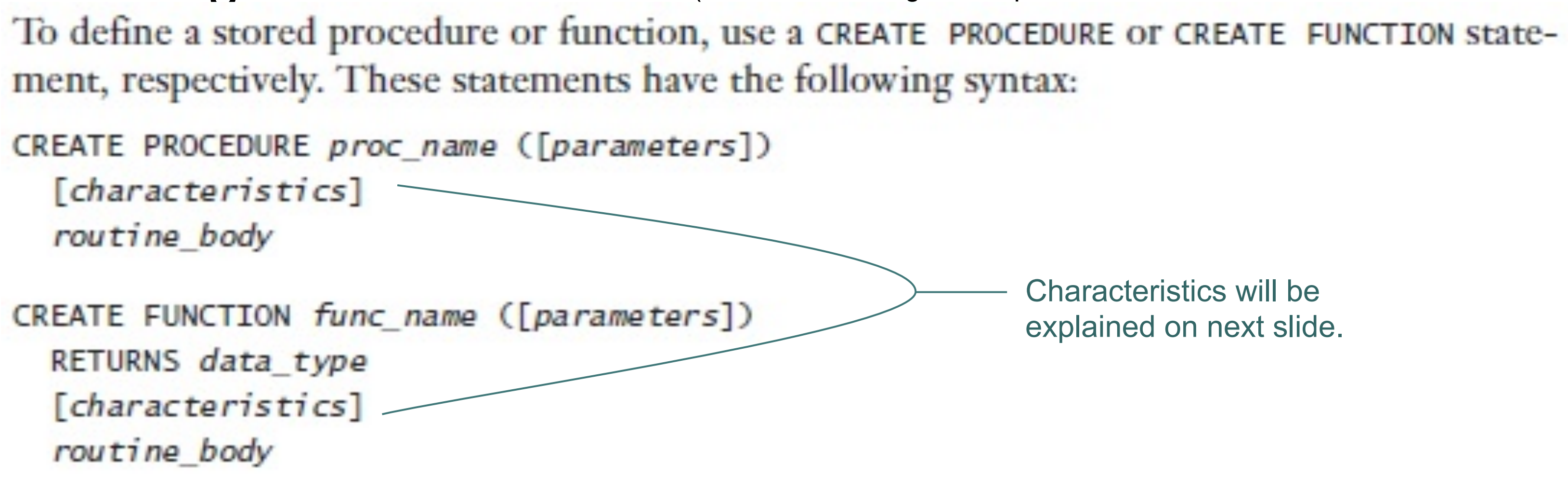
- Stored routine definitions can use compound statements. This definition can be given as a BEGIN/END block that contains multiple statement. Each statement within a block must be terminated by a semicolon character (‘;’).
- If you use the mysql client program, semicolons are ambiguous within routine definitions because mysql itself treats semicolon as a statement terminator. To resolve this issue, mysql supports a **delimiter** command that enables you to change its statement terminator temporarily.
- Example: (the choice of delimiter is up to you. By tradition, // is often used MySQL)


Redefine // as a statement terminator
Redefine ; back as a statement terminator

```
mysql> delimiter //
mysql> CREATE PROCEDURE world_record_count ()
    -> BEGIN
    ->   SELECT 'Country', COUNT(*) FROM Country;
    ->   SELECT 'City', COUNT(*) FROM City;
    ->   SELECT 'CountryLanguage', COUNT(*) FROM CountryLanguage;
    -> END;
    -> //
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Ch 18. Stored Procedures and Functions

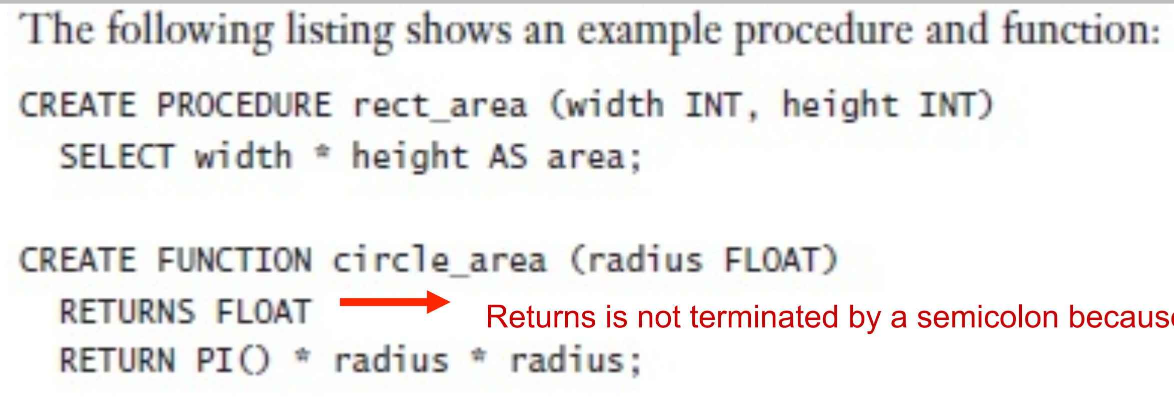
Creating Stored Routines (details of creating stored procedures and functions will be discussed from Slide 19)



To define a stored procedure or function, use a CREATE PROCEDURE or CREATE FUNCTION statement, respectively. These statements have the following syntax:

```
CREATE PROCEDURE proc_name ([parameters])
  [characteristics]
  routine_body

CREATE FUNCTION func_name ([parameters])
  RETURNS data_type
  [characteristics]
  routine_body
```

Characteristics will be explained on next slide.



The following listing shows an example procedure and function:

```
CREATE PROCEDURE rect_area (width INT, height INT)
  SELECT width * height AS area;

CREATE FUNCTION circle_area (radius FLOAT)
  RETURNS FLOAT
  RETURN PI() * radius * radius;
```

Returns is not terminated by a semicolon because it is just a clause, not a statement.

Ch 18. Stored Procedures and Functions

- The characteristics clause is optional and contains one or more of the following values, which can appear in any order.
  - SQL SECURITY {DEFINER | INVOKER} : A stored routine runs either with privileges of the user who created it or the user who invoked it. The choice of which set of privileges to use is controlled by the value of the SQL SECURITY characteristic:
    - A value of DEFINER causes the routine to have the privilege of the users who created it. This is a default value.
    - A value of INVOKER causes the routine to run with the privileges of it invoking user. This means the routine has access to database objects only if that use can already access them otherwise.
  - DETERMINSTIC or NOT DETERMINISTIC : This indicates whether the routine always produces the same result when invoked with a given set of input parameter values. If it does not, the routine is non-deterministic (such as a function that returns a summary of financial data that changes over time). The default value is NOT DETERMINISTIC.
  - Language SQL : This indicates the language in which the routine is written. Currently, the only supported language is SQL, so SQL is the only allowable value for the LANGUAGE characteristic and also the default value.
  - Comment 'String' : This specifies a descriptive string for the routine.

Ch 18. Stored Procedures and Functions

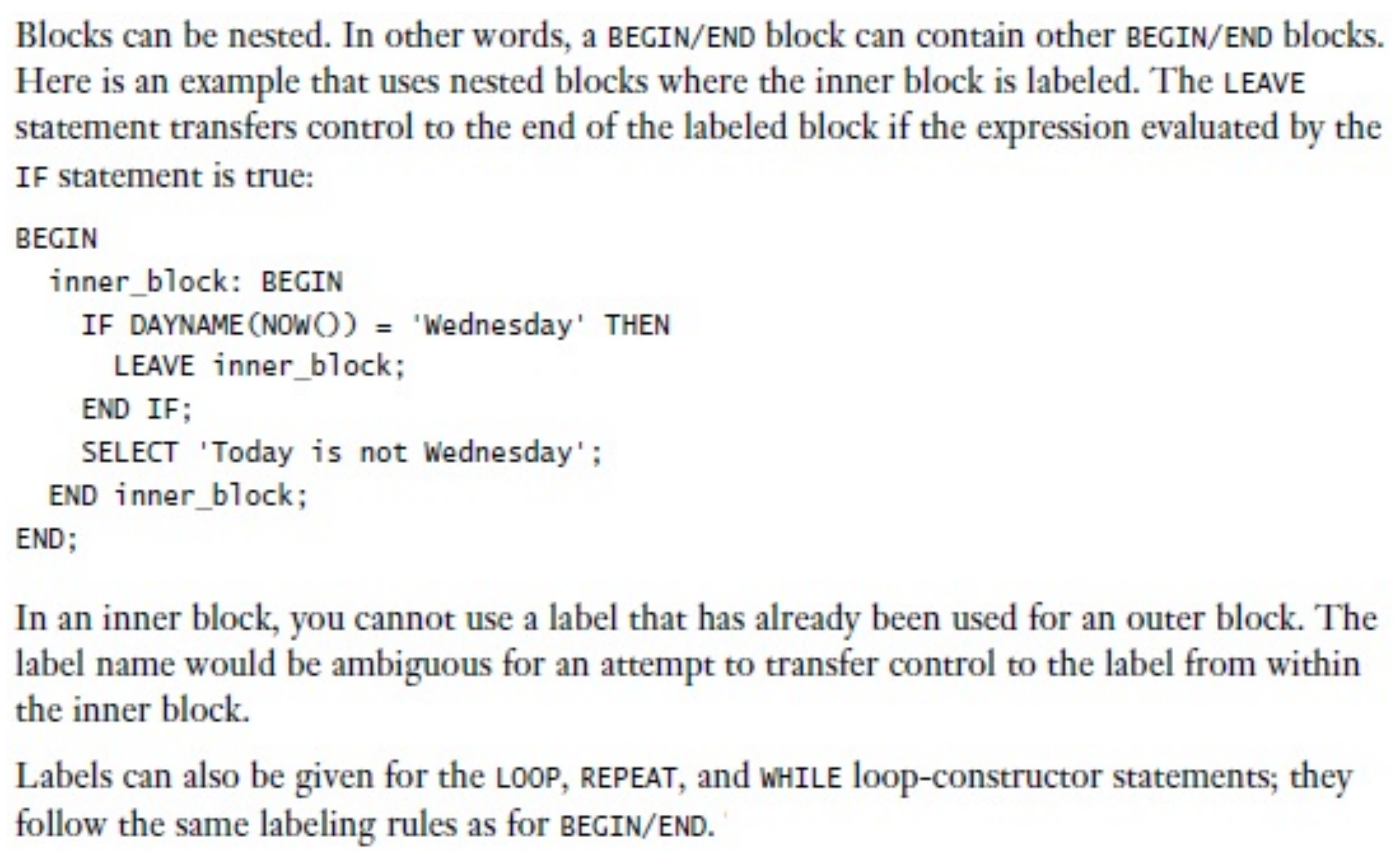
Compound Statements

- For a complex routine, you can use a compound statement that contains other statements. A compound statement begins and ends with the BEGIN and END keywords and create a block. In between BEGIN and END, write the statements that make up the block, each terminated by a semicolon character (‘;’). The BEGIN/END block itself is terminated by a semicolon, but BEGIN is not. Below is an example:

```
mysql> CREATE PROCEDURE world_record_count()
    BEGIN
        SELECT 'Country', COUNT(*) FROM Country;
        SELECT 'City', CoUNT(*) FROM City;
        SELECT 'CountryLanguage', COUNT(*) FROM CountryLanguage;
    END;
```

- A block can be labeled, which is useful when it’s necessary to alter the flow of control. For example, the LEAVE statement can be used to exit a labeled BEGIN/END block. The syntax for labeling a block looks like this : **[*label:*] BEGIN…END [*label*].**
- Labels are optional, but the label at the end can be present only if the label at the beginning is also present, and the end label must have the same name as the beginning label.

Ch 18. Stored Procedures and Functions



Blocks can be nested. In other words, a BEGIN/END block can contain other BEGIN/END blocks. Here is an example that uses nested blocks where the inner block is labeled. The LEAVE statement transfers control to the end of the labeled block if the expression evaluated by the IF statement is true:

```
BEGIN
  inner_block: BEGIN
    IF DAYNAME(NOW()) = 'Wednesday' THEN
      LEAVE inner_block;
    END IF;
    SELECT 'Today is not Wednesday';
  END inner_block;
END;
```

In an inner block, you cannot use a label that has already been used for an outer block. The label name would be ambiguous for an attempt to transfer control to the label from within the inner block.

Labels can also be given for the LOOP, REPEAT, and WHILE loop-constructor statements; they follow the same labeling rules as for BEGIN/END.

Ch 18. Stored Procedures and Functions

STORED PROCEDURE vs. STORED FUNCTION.

1. Stored Function input Arguments Only, Stored Procedure input and output Arguments.
2. Stored Function one return value via the return statement,
   - Stored Procedures send multiple return values via output argument.
   - Stored Procedures can also send back multiple ResultSets to client.
3. Stored Functions can usually be used in SQL Statements, Stored Procedures cannot.

STORED FUNCTIONS (CREATE FUNCTION):

- Only accept input parameters (IN).
- Always return one value.
- Invoked within expressions just as any SQL function might be, like (sum, average, count, etc.)

Ch 18. Stored Procedures and Functions

Syntax

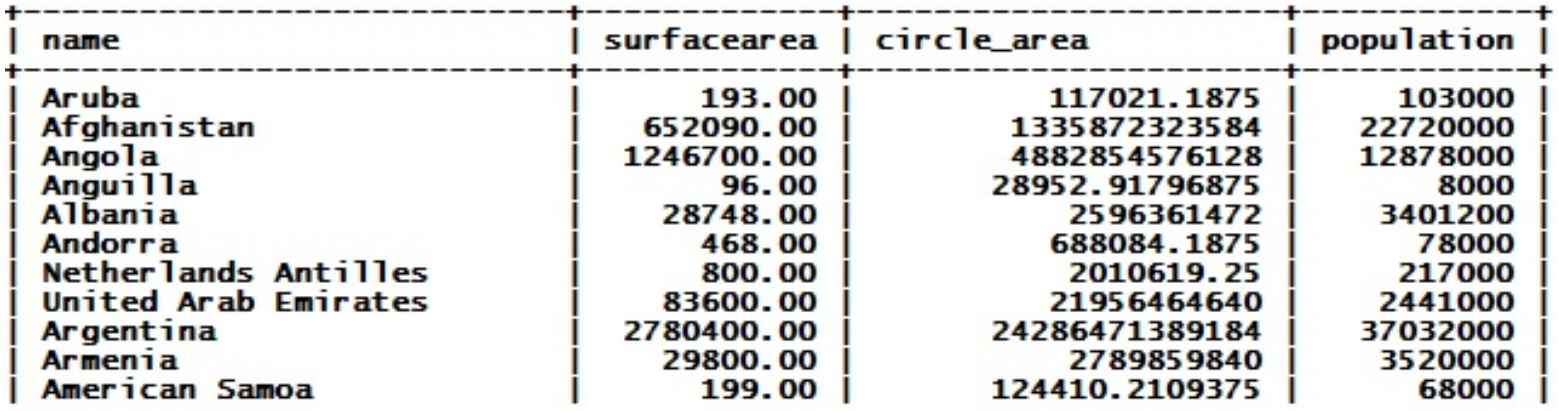
```
mysql> CREATE FUNCTION function_name (argument1 datatype1, Argument2 datatype2, ?)
RETURNS datatype
BEGIN
        Statements;
END;

/* get_balance(p_customer_number): retrieve customer balance.
   @author: rtimlin
   @original: 06-Mar-2001
   @updated:
   @arguments: p_cust_nr Customer number from the Customer Table:
   @version: 1.0

                 **** Modification History  ****

   Date       User       Description
   ==================================================
   3/6/01    rtimlin       Original.
   3/8/01    flast          Added Exception handler.
*/
```

Ch 18. Stored Procedures and Functions

```
CREATE FUNCTION circle_area (radius FLOAT)
   RETURNS FLOAT
   RETURN PI() * radius * radius;
// Comparison with Java/C++, etc.
public float circle_area (float radius)
{ // BEGIN
   return pi() * radius * radius;
} // END;
```

Ch 18. Stored Procedures and Functions

mysql> SELECT name, surfacearea, circle_area(surfacearea) circle_area, population FROM country;

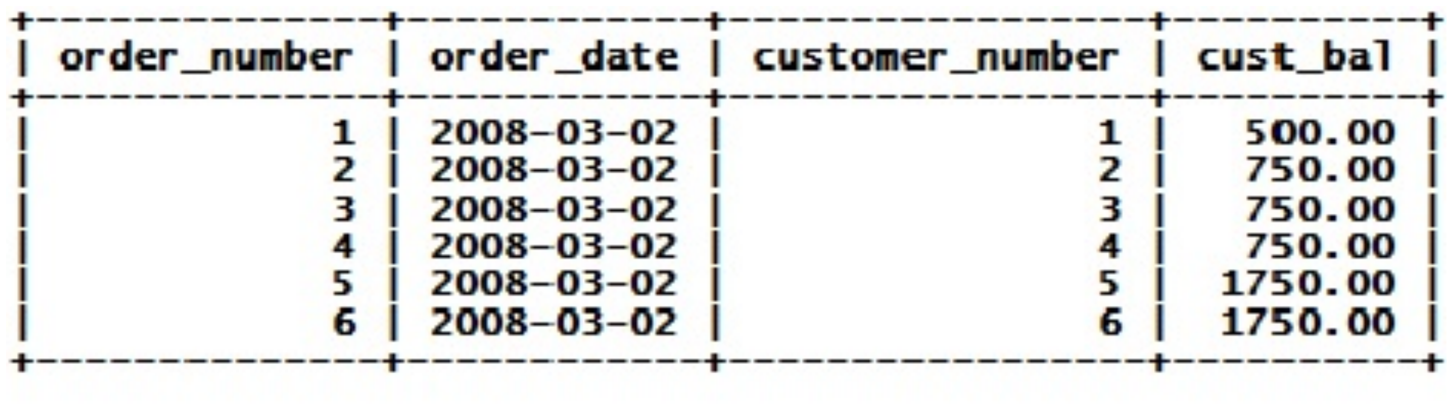


```
+----------------------------+-------------+----------------------+------------+
| name                       | surfacearea | circle_area          | population |
+----------------------------+-------------+----------------------+------------+
| Aruba                      |      193.00 |          117021.1875 |     103000 |
| Afghanistan                |   652090.00 |        1335872323584 |   22720000 |
| Angola                     |  1246700.00 |        4882854576128 |   12878000 |
| Anguilla                   |       96.00 |       28952.91796875 |       8000 |
| Albania                    |    28748.00 |           2596361472 |    3401200 |
| Andorra                    |      468.00 |          688084.1875 |      78000 |
| Netherlands Antilles       |      800.00 |           2010619.25 |     217000 |
| United Arab Emirates       |    83600.00 |          21956464640 |    2441000 |
| Argentina                  |  2780400.00 |       24286471389184 |   37032000 |
| Armenia                    |    29800.00 |           2789859840 |    3520000 |
| American Samoa             |      199.00 |       124410.2109375 |      68000 |
```

```
mysql> DROP FUNCTION get_balance;
mysql> DELIMITER //
mysql> CREATE FUNCTION get_balance (p_customer_number INTEGER) RETURNS DECIMAL(10,2)
BEGIN
    DECLARE v_balance DECIMAL(10,2);
     SELECT balance INTO v_balance FROM customer
    WHERE customer_number = p_customer_number;
     RETURN v_balance;
END;
//
mysql> DELIMITER ;
```

Ch 18. Stored Procedures and Functions

```
mysql> SELECT *, get_balance(customer_number) as cust_bal FROM orders;
```



```
+--------------+------------+-----------------+----------+
| order_number | order_date | customer_number | cust_bal |
+--------------+------------+-----------------+----------+
|            1 | 2008-03-02 |               1 |   500.00 |
|            2 | 2008-03-02 |               2 |   750.00 |
|            3 | 2008-03-02 |               3 |   750.00 |
|            4 | 2008-03-02 |               4 |   750.00 |
|            5 | 2008-03-02 |               5 |  1750.00 |
|            6 | 2008-03-02 |               6 |  1750.00 |
+--------------+------------+-----------------+----------+
```

```
mysql> CREATE FUNCTION tax (cost DECIMAL(10,2), tax_rate DECIMAL(10,2))
    RETURNS DECIMAL(10,4)
    RETURN cost * tax_rate;
```

***Note the indentation above, failure to indent for readability will result in one full letter grade deduction and your work being returned to you to fix. Also I will not help you with any program that is NOT readable.

Ch 18. Stored Procedures and Functions

STORED PROCEDURES (CREATE PROCEDURE):

- Invoked in client using the CALL command.
- Can have IN, OUT, IN OUT modes of parameters.
- The IN qualifier is used for arguments for which values must be specified when calling the procedure. In the prior CREATE FUNCTION example, v_customer_number is an IN parameter and therefor must be specified.
- The OUT qualifier signifies that the procedure passes a value back to caller through this argument.
- The IN OUT qualifier signifies that this argument is both an IN and an OUT parameter.

Syntax:

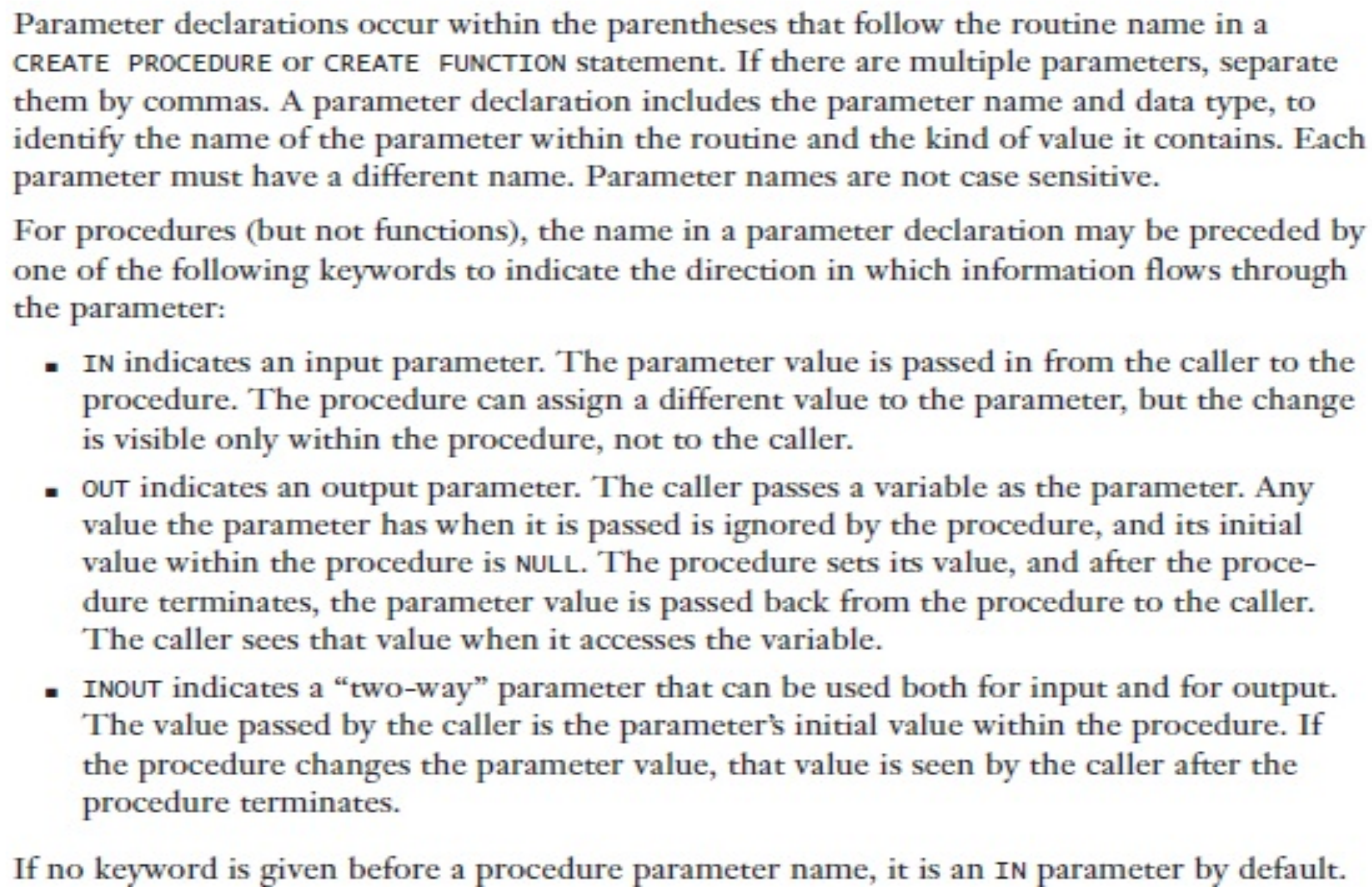
```
CREATE PROCEDURE procedure ([IN|OUT|INOUT] argument1 datatype1, [IN|OUT|INOUT] argument2 datatype2, ? )
BEGIN
        Statements;
END;
```

Example:

```
DROP PROCEDURE world_record_count;
delimiter $$
CREATE PROCEDURE world_record_count()
BEGIN
      SELECT 'Country', COUNT(*) FROM Country;
      SELECT 'City', CoUNT(*) FROM City;
      SELECT 'CountryLanguage', COUNT(*) FROM CountryLanguage;
END;
$$
Delimiter ;
```

Ch 18. Stored Procedures and Functions

Declaring Parameters



Parameter declarations occur within the parentheses that follow the routine name in a `CREATE PROCEDURE` or `CREATE FUNCTION` statement. If there are multiple parameters, separate them by commas. A parameter declaration includes the parameter name and data type, to identify the name of the parameter within the routine and the kind of value it contains. Each parameter must have a different name. Parameter names are not case sensitive.

For procedures (but not functions), the name in a parameter declaration may be preceded by one of the following keywords to indicate the direction in which information flows through the parameter:

- `IN` indicates an input parameter. The parameter value is passed in from the caller to the procedure. The procedure can assign a different value to the parameter, but the change is visible only within the procedure, not to the caller.
- `OUT` indicates an output parameter. The caller passes a variable as the parameter. Any value the parameter has when it is passed is ignored by the procedure, and its initial value within the procedure is `NULL`. The procedure sets its value, and after the procedure terminates, the parameter value is passed back from the procedure to the caller. The caller sees that value when it accesses the variable.
- `INOUT` indicates a "two-way" parameter that can be used both for input and for output. The value passed by the caller is the parameter's initial value within the procedure. If the procedure changes the parameter value, that value is seen by the caller after the procedure terminates.

If no keyword is given before a procedure parameter name, it is an `IN` parameter by default.

Ch 18. Stored Procedures and Functions

```
CREATE PROCEDURE param_test (
        IN        p_in          INT,
        OUT       p_out         INT,
        INOUT     p_inout       INT)
BEGIN
        SELECT p_in, p_out, p_inout;
        SET p_in=100, p_out=200, p_inout=300;
END;

mysql> SET @v_in = 0, @v_out=0, @v_inout=0;

mysql> CALL param_test(@v_in, @v_out, @v_inout);
+------+---------+------------+
| p_in | p_ou t | p_inout  |
+------+---------+------------+
|     0 |  NULL |       0     |
+------+--------+-------------+

mysql> SELECT @v_in, @v_out, @v_inout;
+---------+------------+---------------+
| @v_in  | @v_out | @v_inout   |
+---------+------------+---------------+
| 0          | 200        | 300             |
+---------+-----------+----------------+
```

- **Parameters to stored routines need not be passed as user variables. They can be given as constants or expressions as well. However, for OUT or INOUT procedure parameters, if you do not pass a variable, the value passed back from a procedure will not be accessible**

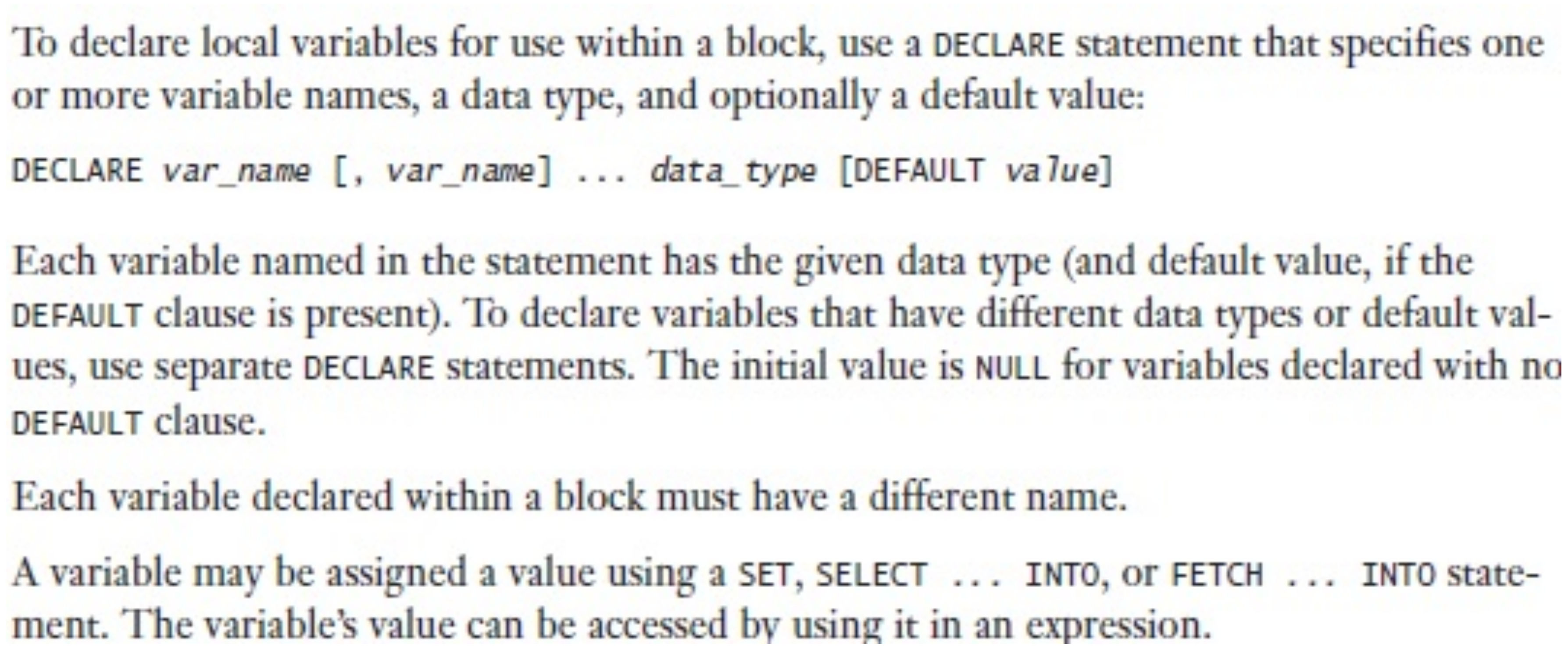
Ch 18. Stored Procedures and Functions

The DECLARE Statement

- The DECLARE statement is used for declaring several types of items in stored routines:
  - Local Variables.
  - Conditions such as warnings or errors (exceptions).
  - Handlers for conditions.
  - Cursors for access data row by row.
- DECLARE statement can be used only within a BEGIN/END block and must appear in the block before any other statements. IF used to declare several types of items within a block, the DECLARE statements must appear in a particular order: You must declare variables and conditions first, then cursors, and finally handlers.
- Each variable declared within a block must have a different name. This restriction also applies to declarations for conditions and for cursors.
- Variables, conditions, handlers and cursors created by DECLARE statements are local to the block. They are valid only within the block (or any nested blocks)

Ch 18. Stored Procedures and Functions

Variables in Stored Routines



To declare local variables for use within a block, use a DECLARE statement that specifies one or more variable names, a data type, and optionally a default value:

```
DECLARE var_name [, var_name] ... data_type [DEFAULT value]
```

Each variable named in the statement has the given data type (and default value, if the DEFAULT clause is present). To declare variables that have different data types or default values, use separate DECLARE statements. The initial value is NULL for variables declared with no DEFAULT clause.

Each variable declared within a block must have a different name.

A variable may be assigned a value using a SET, SELECT ... INTO, or FETCH ... INTO statement. The variable's value can be accessed by using it in an expression.

Ch 18. Stored Procedures and Functions

Assigning Variables Using SELECT

```
SELECT column1, column2
INTO    v_variable1, v_variable2
FROM    table_name
WHERE   condition;

delimiter //
CREATE PROCEDURE get_Country_data (
    IN      p_code      CHAR(3),
    OUT     p_name      VARCHAR(50),
    OUT     p_pop       INT)
BEGIN
    DECLARE name_var VARCHAR(50);
    DECLARE pop_var INT;

    SELECT Name, Population INTO name_var, pop_var
        FROM Country WHERE code = p_code;

    SET p_name = name_var, p_pop = pop_var;
END;
//
delimiter ;
```

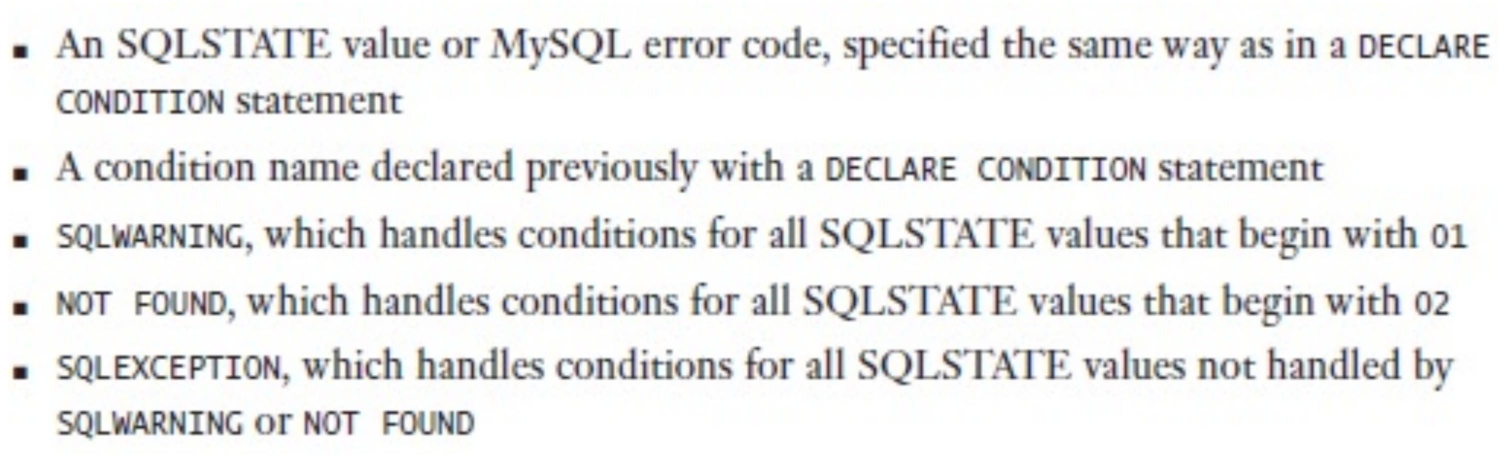
Ch 18. Stored Procedures and Functions

```
mysql> set @v_name = '';
Query OK, 0 rows affected (0.00 sec)

mysql> set @v_pop=0;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL get_Country_data ('USA', @v_name, @v_pop);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @v_name, @v_pop;
+-------------------+---------------+
| @v_name           | @v_pop        |
+-------------------+---------------+
| United States     | 278357000     |
+-------------------+---------------+
```

Ch 18. Stored Procedures and Functions

Conditions and Handlers

- A handler has a name and a statement to be executed upon occurrence of a given condition such as a warning or an error. Handlers commonly are used for detecting problems and dealing with them in a more appropriate way than simply having the routine terminate with an error.
- Each condition associated with a handler must be one of the following:



- An SQLSTATE value or MySQL error code, specified the same way as in a `DECLARE CONDITION` statement
- A condition name declared previously with a `DECLARE CONDITION` statement
- `SQLWARNING`, which handles conditions for all SQLSTATE values that begin with `01`
- `NOT FOUND`, which handles conditions for all SQLSTATE values that begin with `02`
- `SQLEXCEPTION`, which handles conditions for all SQLSTATE values not handled by `SQLWARNING` or `NOT FOUND`

Ch 18. Stored Procedures and Functions

- Named conditions and handlers are both declared with DECLARE statement. Conditions must be declared along with variable before any cursor or handler declarations. Handler declarations must follow declarations for variables, conditions, and cursors.
- To name a condition, use DECLARE CONDITION statement:

DECLARE *condition_name* **CONDITION FOR** *condition type*;

- A condition type must be an SQLSTATE value, specified as SQLSTATE (or SQLSTATE VALUE) followed by a five-character string literal. For example:

DECLARE null_not_allowed CONDITION FOR SQLSTATE ‘23000‘;

- The DECLARE HANDLER statement creates a handler for one or more conditions and asscoiates them with an SQL statement that will be executed should any of the conditions occur when the routine is run:

DECLARE *handler_type* **HANDLER FOR**

condition_type [, *Condition_type*]…

statement

Ch 18. Stored Procedures and Functions

Handlers can be single statement or compound, i.e. between BEGIN & END;

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'  SET exit_loop = 1;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
BEGIN
   Statements;
END;
```

$\longrightarrow$ Single statement

$\}\longrightarrow$ Compound statement

More Example:

```
mysql> CREATE TABLE unique_names (name VARCHAR(50) NOT NULL PRIMARY KEY);
mysql> CREATE TABLE dup_names (name VARCHAR(50));

mysql> delimiter //

mysql> CREATE PROCEDURE add_name (name_param CHAR(20))
BEGIN
DECLARE EXIT HANDLER FOR SQLSTATE '23000'
BEGIN
INSERT INTO dup_names (name) VALUES (name_param);
SELECT 'duplicate key found, inserted into dup_names' AS result;
END;

INSERT INTO unique_names (name) VALUES (name_param);
SELECT 'row inserted successfully into unique_names' AS Result;
END;
//
mysql> delimiter ;
```

Ch 18. Stored Procedures and Functions

- To ignore a condition, declare a CONDITION handler for it and associate it with an empty block;

DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;

Cursors

- A cursor enables you to access a result set one row at a time. Because of this row orientation cursors often are used in loops that fetch and process a row within each iteration of the loop.
- The cursor implementation in MySQL has the following properties: It provides for real only cursors; they can not be used to modify tables. Cursors also only advance through a result row by row. They are not scrollable.
- Creating a cursor:
- Declare the cursor.
- Open the cursor.
- Fetch data from the cursor.
- Close the cursor.

Ch 18. Stored Procedures and Functions

Declaring Cursors, Syntax:

```
DECLARE cursor_name CURSOR FOR
        SELECT columns
        FROM tables
        WHERE condition;
```

Notes:

- Do not include the INTO clause in the SELECT.
- Declare variables before cursors.

Opening Cursors, Syntax:

```
OPEN cursor_name;
```

Note: This is done in the BEGIN section and is used to prepare the cursor for execution.

Fetching Data from the Cursor, Syntax:

```
FETCH cursor_name INTO variable1, variable2, etc.;
```

Notes:

- Include the same number of variables as is included in the SELECT clause of the cursor.
- Variables are mapped to columns in the SELECT in the order they are listed.

Closing Cursors, Syntax:

```
CLOSE cursor_name;
```

Notes:

- Close the cursor after all the rows are processed.
- You can't fetch data from the cursor once you have closed it.

Ch 18. Stored Procedures and Functions

EXAMPLE 1:

```
delimiter //
CREATE PROCEDURE cursor_example ()
BEGIN
    DECLARE row_count INT DEFAULT 0;
    DECLARE code_var CHAR(3);
    DECLARE name_var CHAR(52);
    DECLARE c CURSOR FOR  -- Cursor is like a Java ResultSet
            SELECT Code, Name FROM Country WHERE Continent = 'Africa';
    OPEN c;
    -- ResultSet rst = conn.executeQuery("SELECT Code, Name FROM Country WHERE Continent = 'Africa'");
    BEGIN
            DECLARE EXIT HANDLER FOR SQLSTATE '02000' BEGIN END;
            //NOTE: Exits the nearest BEGIN/END block, not the LOOP underneath it.
            LOOP
                    FETCH c INTO code_var, name_var;
                    SET row_count = row_count +1;
            END LOOP;
    END;
    CLOSE c;
    SELECT 'number of rows fetched =', row_count;
END;
//
delimiter ;
```

Ch 18. Stored Procedures and Functions

EXAMPLE 2:

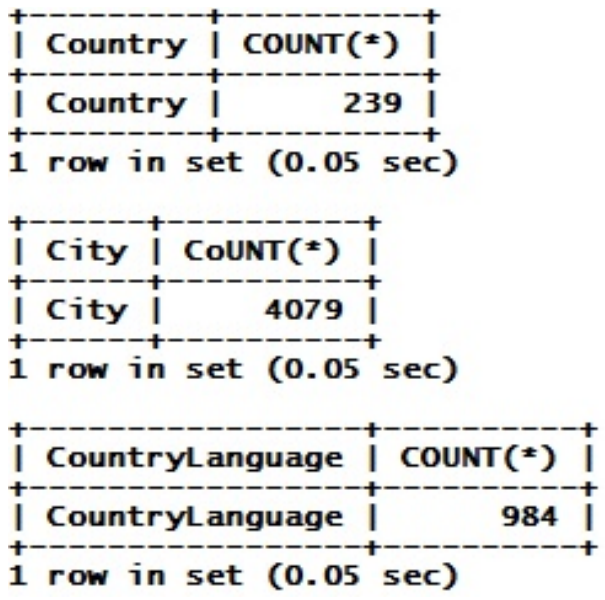
```
delimiter //
CREATE PROCEDURE cursor_example2 ()
BEGIN
    DECLARE row_count INT DEFAULT 0;
    DECLARE exit_flag INT DEFAULT 0;
    DECLARE code_var CHAR(3);
    DECLARE name_var CHAR(52);
    DECLARE rst CURSOR FOR  -- Like a Java ResultSet
            SELECT Code, Name FROM Country WHERE Continent = 'Africa';

    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET exit_flag=1;
    -- ResultSet rst = conn.executeQuery("SELECT Code, Name FROM Country WHERE Continent = 'Africa'");

    OPEN rst;
    fetch_loop: LOOP
            /* Next two likes equate to:
            if (!rst.next())
                    break;
            */
            FETCH rst INTO code_var, name_var;
            IF exit_flag THEN
                    LEAVE fetch_loop;
            END IF;
            SET row_count = row_count +1;
    END LOOP;
    CLOSE rst;  -- rst.close();
    SELECT 'number of rows fetched =', row_count;
END;
//
delimiter ;
```

Ch 18. Stored Procedures and Functions

Retrieving Multiple Result Sets

- A MySQL extension to procedures is that SELECT statements can be executed to generate result sets that are return directly to the client with no intermediate processing. The client retrieves the results as thought it has executed the SELECT statement itself. This extension does NOT apply to stored functions.

mysql> call world_record_count();



```
+---------+----------+
| Country | COUNT(*) |
+---------+----------+
| Country |      239 |
+---------+----------+
1 row in set (0.05 sec)

+------+----------+
| City | COUNT(*) |
+------+----------+
| City |     4079 |
+------+----------+
1 row in set (0.05 sec)

+-----------------+----------+
| CountryLanguage | COUNT(*) |
+-----------------+----------+
| CountryLanguage |      984 |
+-----------------+----------+
1 row in set (0.05 sec)
```

Ch 18. Stored Procedures and Functions

Flow Control

- Compound statement syntax includes statements that allow for conditional testing and for creating looping structures:
- IF and CASE perform conditional testing
- LOOP. REPEAT and WHILE create loops. LOOP iterates unconditionally, whereas REPEAT and WHILE include a clause that tests whether the loop should continue or terminate.

Please refer to the following link for details and examples.

http://www.timlin.net/csm/cis363/mysql18a.htm

Ch 18. Stored Procedures and Functions

Altering Stored Routines

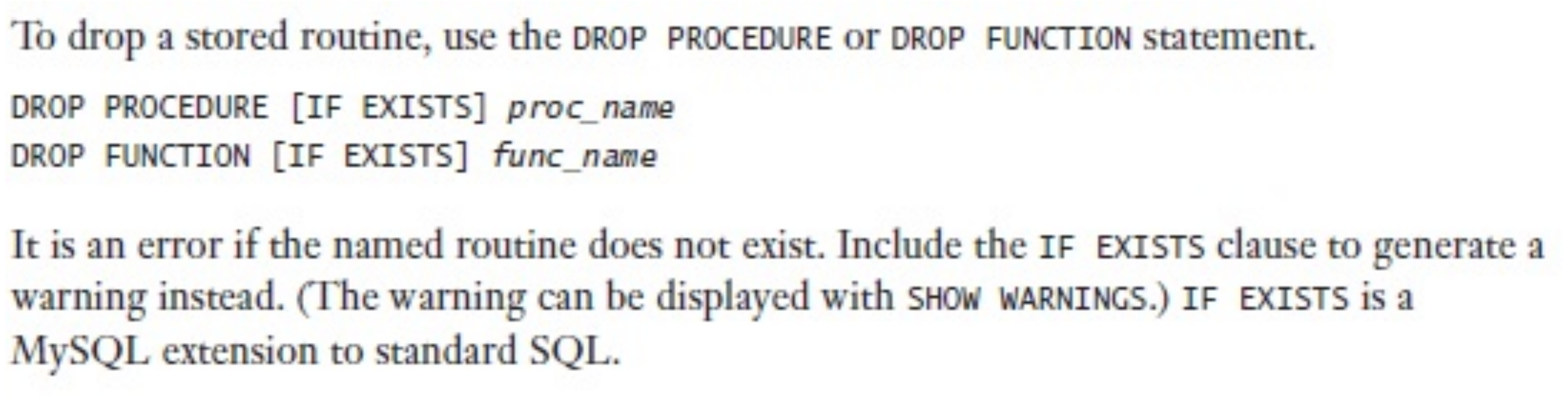
- The Alter PROCEDURE or ALTER FUNCTION statement can be used to alter some of the characteristics of a stored routine:

```
ALTER PROCEDURE proc_name [characteristics]
ALTER FUNCTION func_name [characteristics]
```

- The allowable characteristics for these statements are SQL SECURITY and COMMENT. These statements can NOT be used to alter other aspects of routine definitions such as the parameter declarations or the body. To do that, you must drop the routine and the create it again with new definition.

Ch 18. Stored Procedures and Functions

Dropping Stored Routines.



To drop a stored routine, use the DROP PROCEDURE or DROP FUNCTION statement.

```
DROP PROCEDURE [IF EXISTS] proc_name
DROP FUNCTION [IF EXISTS] func_name
```

It is an error if the named routine does not exist. Include the IF EXISTS clause to generate a warning instead. (The warning can be displayed with SHOW WARNINGS.) IF EXISTS is a MySQL extension to standard SQL.

Ch 18. Stored Procedures and Functions

Invoking Stored routines

- To invoke a procedure, use CALL statement. This is a separate statement; a procedure cannot be invoked as part of an expression.
- To invoke a function, invoke it in an expression. It returns as a single value that is used in evaluating the expression, just as for a built-in function.