MYSQL Stored Procedure

S. Ranbir Singh

Stored Procedures

Up until now, all of our data retrieval has been accomplished with a single statement. Even the use of subqueries was accomplished by combining two SELECTs into a single statement. We're now going to discuss a new scenario in which multiple statements can be saved into a single object known as a **stored procedure**.

A **Stored Procedure** is a set of SQL statements, compiled and stored as a single database object *for repeated use.*

- It is used to get information from the database or change data in the database
- It is used by **application** programs (along with **views**)
- It can use **zero** or **more parameters**
- It is run using a **CALL** statement (in MySQL) with the procedure name and any parameter values
- It is built using a **CREATE PROCEDURE** statement.

Stored Procedures

In broad terms, there are two general reasons why you might want to use stored procedures:

- To save **multiple** SQL statements in a **single** procedure
- To use **parameters** in conjunction with your SQL statements

Stored procedures can, in fact, consist of a single SQL statement and contain no parameters.

But the real value of stored procedures becomes evident when they contain multiple statements or parameters.

This is something that relates directly to the issue of how to best retrieve data from a database.

Stored Procedures

Basically, the ability to store multiple statements in a procedure means that you can create complex logic and execute it all at once as a single transaction.

For example, you might have a business requirement to take an incoming order from a customer and quickly evaluate it before accepting it from the customer.

This procedure *might* involve:

- **checking** to make sure that the items are in **stock**
- **verifying** that the customer has a good **credit rating**
- **getting an initial estimate** as to when it can be **shipped**

This situation would require multiple SQL statements with some added logic to determine what kind of message to return if all were not well with the order. All of that logic could be placed into a single stored procedure, which would enhance the modularity of the system.

With everything in one procedure, that logic could be executed from any calling program, and it would always return the same result.

Stored Procedures

Why Use Stored Procedures?

One of the most beneficial reasons to use stored procedures is the added layer of **security** that can be placed on the database from the calling application.

If the user account created for the application or web site is configured with permissions only then the underlying tables cannot be accessed directly by the user account. This helps prevent hacking directly into the database tables.

The risk of a hacker using the user account to run a stored procedure that has been written by you is far safer than having the user account have full insert, update and delete authority on the tables directly.

Stored Procedures

Benefits of Stored Procedures

- **Modular Programming** – You can write a stored procedure once, then call it from multiple places in your application.

- **Performance -** Stored procedures provide faster code execution and reduce network traffic.
  - Faster execution: Stored procedures are parsed and optimized as soon as they are created and the stored procedure is stored in memory. This means that it will execute a lot faster than sending many lines of SQL code from your application to the SQL Server. Doing that requires SQL Server to compile and optimze your SQL code every time it runs.
  - Reduced network traffic: If you send many lines of SQL code over the network to your SQL Server, this will impact on network performance. This is especially true if you have hundreds of lines of SQL code and/or you have lots of activity on your application. Running the code on the SQL Server (as a stored procedure) eliminates the need to send this code over the network. The only network traffic will be the parameters supplied and the results of any query.

- **Security** - Users can execute a stored procedure without needing to execute any of the statements directly. Therefore, a stored procedure can provide advanced database functionality for users who wouldn't normally have access to these tasks, but this functionality is made available in a tightly controlled way.

Stored Procedures

Disadvantages of Stored Procedures

- Increased **load** on the database server — most of the work is done on the **server side**, and less on the client side.

- There’s a decent **learning curve**. You’ll need to learn not only the syntax of SQL statements in order to write stored procedures, but the particular "**dialect**" of the **DBMS** managing them (e.g., SSQL Server T-SQL vs. MySQL vs Oracle vs DBs)

- You are repeating the logic of your application in two different places: your **server code** and the **stored procedures code**, making things a bit more difficult to maintain.

- **Migrating** to a different **database management system** (MySQL, SQL Server, Oracle, DB2, etc) may potentially be more difficult.

Stored Procedures

MySQL and Stored Procedures

MySQL is known as the most popular open source RDBMS which is widely used by both community and enterprise.

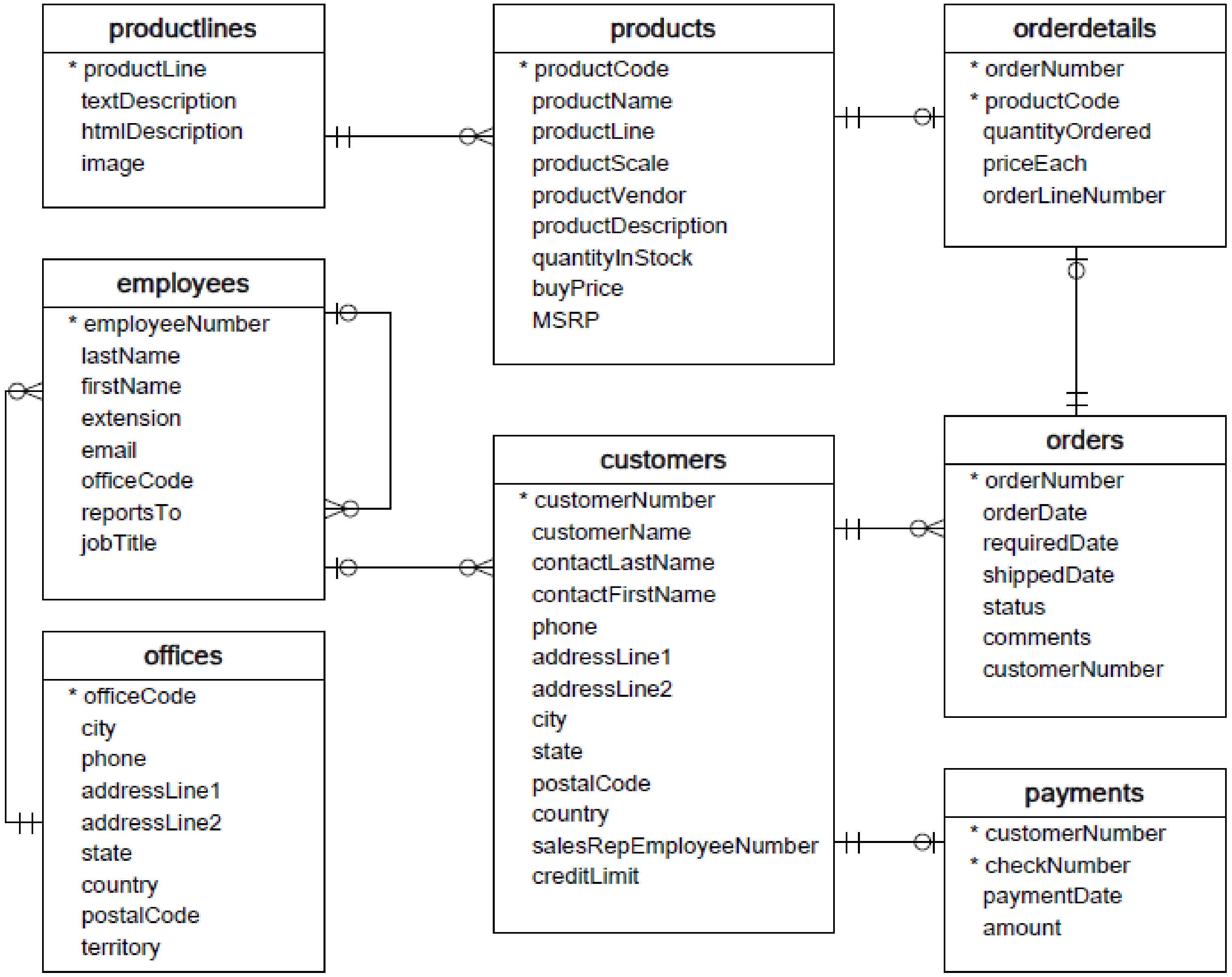
However during the first decade of its existence, it did **not** support **stored procedures**, **triggers**, **events**, etc.

Since MySQL version 5.0 (release in 2009) , those features have been added to MySQL database engine to make it become flexible and powerful.

What follows is a look at stored procedures and how they are created and called in **MySQL** (we will look at stored procedures in **Microsoft SQL Server** on *another* day, after we've had some time to first learn a few of its operating nuances).

https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx
https://www.w3resource.com/mysql/mysql-procedure.php

Example Database

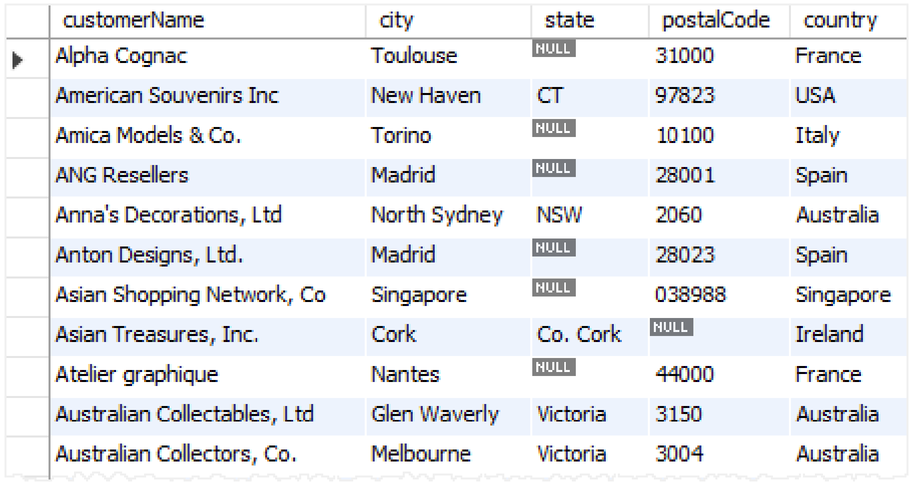


Stored Procedures

The following SELECT statement returns all rows in the table customers from the sample

```
SELECT
    customerName,
    city,
    state,
    postalCode,
    country
FROM
    customers
ORDER BY customerName;
```

This picture shows the partial output of the query:



customerName	city	state	postalCode	country
Alpha Cognac	Toulouse	NULL	31000	France
American Souvenirs Inc	New Haven	CT	97823	USA
Amica Models & Co.	Torino	NULL	10100	Italy
ANG Resellers	Madrid	NULL	28001	Spain
Anna's Decorations, Ltd	North Sydney	NSW	2060	Australia
Anton Designs, Ltd.	Madrid	NULL	28023	Spain
Asian Shopping Network, Co	Singapore	NULL	038988	Singapore
Asian Treasures, Inc.	Cork	Co. Cork	NULL	Ireland
Atelier graphique	Nantes	NULL	44000	France
Australian Collectables, Ltd	Glen Waverly	Victoria	3150	Australia
Australian Collectors, Co.	Melbourne	Victoria	3004	Australia

Stored Procedures

To create a new stored procedure, you use the CREATE PROCEDURE statement. Here is the basic syntax of the CREATE PROCEDURE statement:

```
CREATE PROCEDURE procedure_name(parameter_list)
BEGIN
   statements;
END //
```

In this syntax

- First, specify the name of the stored procedure that you want to create after the CREATE PROCEDURE keywords.
- Second, specify a list of comma-separated parameters for the stored procedure in parentheses after the procedure name.
- Third, write the procedural code between the BEGIN END block.

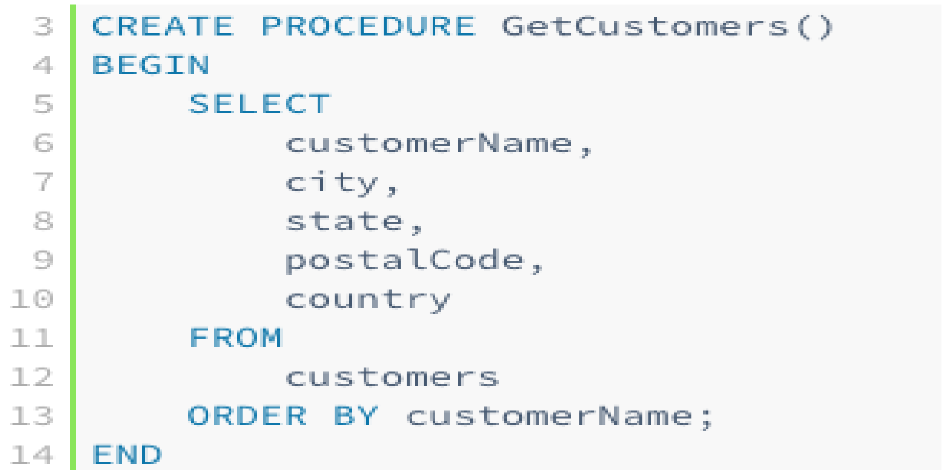
To execute a stored procedure, you use the CALL statement:

```
CALL stored_procedure_name(argument_list);
```

Stored Procedures

Creating a MySQL Stored Procedure

First off, let's look at some attempted stored procedure code:



```
 3  CREATE PROCEDURE GetCustomers()
 4  BEGIN
 5      SELECT
 6          customerName,
 7          city,
 8          state,
 9          postalCode,
10          country
11      FROM
12          customers
13      ORDER BY customerName;
14  END
```

Pretty easy, right? But it causes this error:

```
#1064 – You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near '' at
line 3
```

Okay, so what's the simple fix? The next section about creating stored procedures in MySQL should make everything clear.

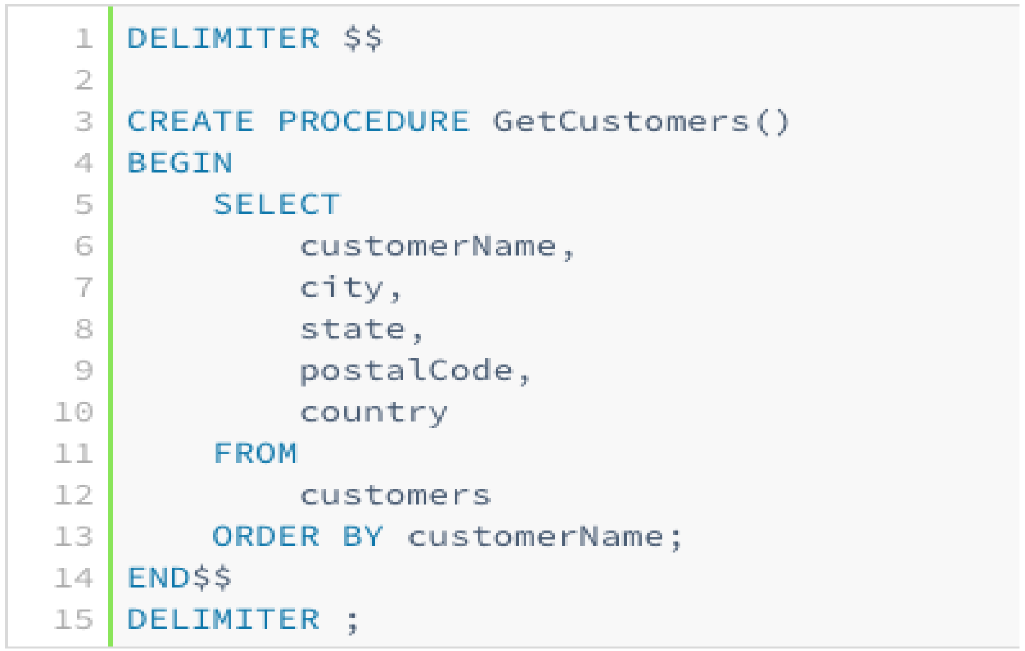
Stored Procedures

Changing the Delimiter

The delimiter is the character or string of characters that you'll use to tell the MySQL client that you've finished typing in an SQL statement. For ages, the delimiter has always been a **semicolon** (**;**). That, however, causes problems, because, in a stored procedure, one can have many statements, and each must end with a **semicolon**. What one can do is to change the delimiter to something else, something *other* than a semicolon.

In this brief overview I'll be using **$$** (you can use anything you want, **//** for example)

Creating a Stored Procedure



```
DELIMITER $$

CREATE PROCEDURE GetCustomers()
BEGIN
    SELECT
        customerName,
        city,
        state,
        postalCode,
        country
    FROM
        customers
    ORDER BY customerName;
END$$
DELIMITER ;
```

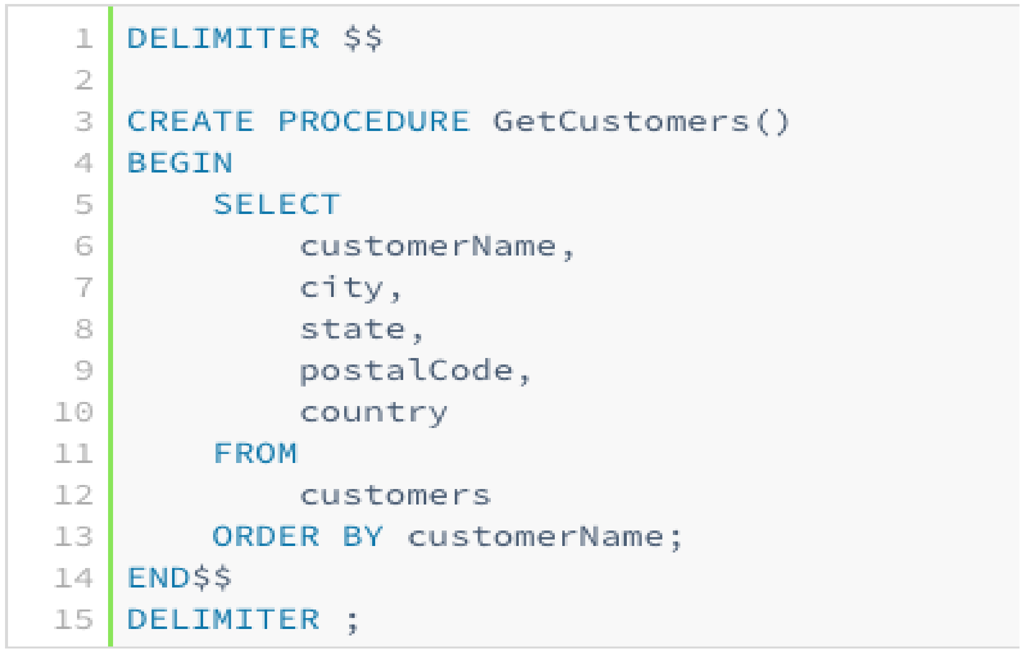
Stored Procedures

Changing the Delimiter

The delimiter is the character or string of characters that you'll use to tell the MySQL client that you've finished typing in an SQL statement. For ages, the delimiter has always been a **semicolon** (;). That, however, causes problems, because, in a stored procedure, one can have many statements, and each must end with a **semicolon**. What one can do is to change the delimiter to something else, something *other* than a semicolon.

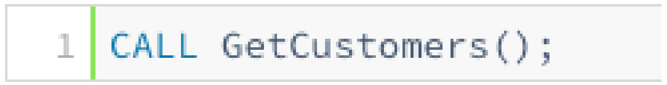
In this brief overview I'll be using **$$** (you can use anything you want, **//** for example)

Creating a Stored Procedure



```
DELIMITER $$

CREATE PROCEDURE GetCustomers()
BEGIN
    SELECT
        customerName,
        city,
        state,
        postalCode,
        country
    FROM
        customers
    ORDER BY customerName;
END$$
DELIMITER ;
```

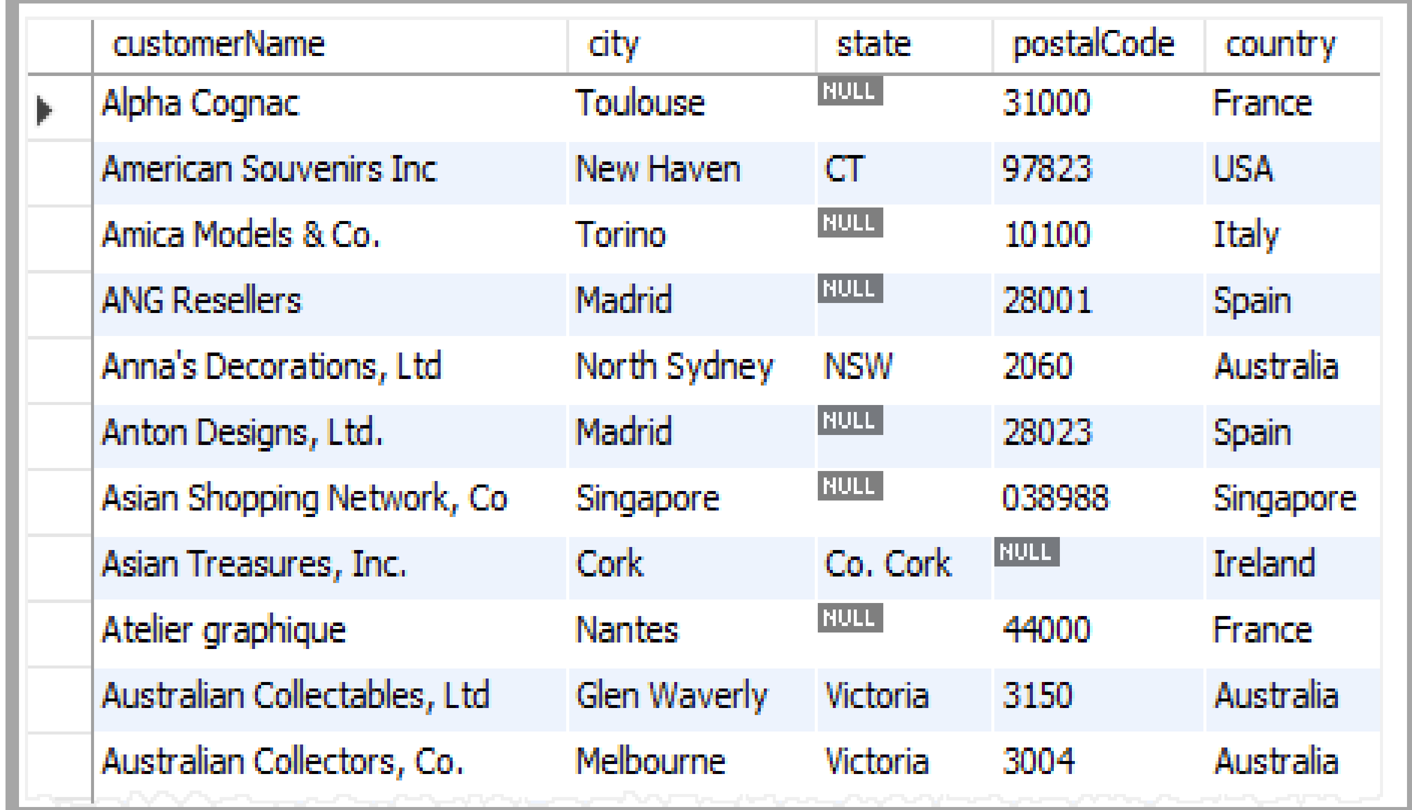
Call Procedure



```
CALL GetCustomers();
```

Stored Procedures

Changing the Delimiter



	customerName	city	state	postalCode	country
▶	Alpha Cognac	Toulouse	NULL	31000	France
	American Souvenirs Inc	New Haven	CT	97823	USA
	Amica Models & Co.	Torino	NULL	10100	Italy
	ANG Resellers	Madrid	NULL	28001	Spain
	Anna's Decorations, Ltd	North Sydney	NSW	2060	Australia
	Anton Designs, Ltd.	Madrid	NULL	28023	Spain
	Asian Shopping Network, Co	Singapore	NULL	038988	Singapore
	Asian Treasures, Inc.	Cork	Co. Cork	NULL	Ireland
	Atelier graphique	Nantes	NULL	44000	France
	Australian Collectables, Ltd	Glen Waverly	Victoria	3150	Australia
	Australian Collectors, Co.	Melbourne	Victoria	3004	Australia

u'll use to tell the MySQL client
he delimiter has always been a
n a stored procedure, one can
n. What one can do is to change
icolon.

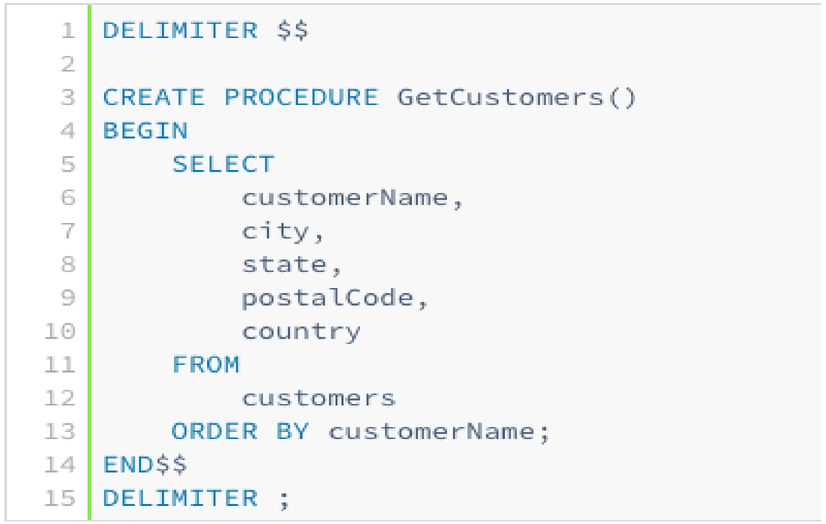
g you want, **//** for example)

Call Procedure

```
CALL GetCustomers();
```

```
        customers
    ORDER BY customerName;
END$$
DELIMITER ;
```

Stored Procedures



```
DELIMITER $$

CREATE PROCEDURE GetCustomers()
BEGIN
    SELECT
        customerName,
        city,
        state,
        postalCode,
        country
    FROM
        customers
    ORDER BY customerName;
END$$
DELIMITER ;
```

Let's examine the stored procedure in a greater detail:

The first command is **DELIMITER $$**, which is **not** related to the stored procedure syntax. The DELIMITER statement changes the standard delimiter which is **semicolon** (**;**) to another. In this case, the delimiter is changed from the semicolon (;) to double-slashes **$$**. Why do we have to change the delimiter? because we want to pass the stored procedure to the server as a whole instead of letting *MySQL* interpret each statement one at a time when we type. Following the **END** keyword, we use delimiter **$$** to indicate the end of the stored procedure. The last command ***DELIMITER ;*** changes the delimiter back to the semicolon (;).

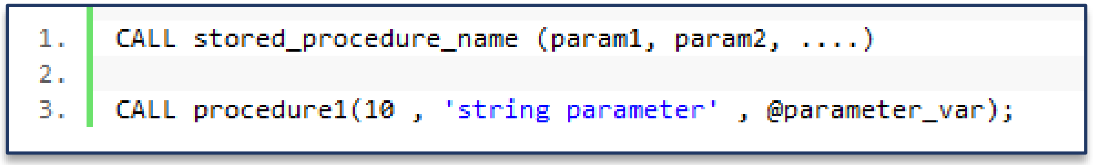
The **CREATE PROCEDURE** statement is used to create a new stored procedure. You can specify the name of stored procedure after the **CREATE PROCEDURE** statement. In this case, the name of the stored procedure is ***GetCustomers***. Do not forget the **parenthesis ( )** after the name of the store procedure or you will get an error message.

Everything inside a pair of keyword **BEGIN** and **END** is called stored procedure's **body**. You can put the declarative SQL code inside the stored procedure's body to handle business logic. In the store procedure we used simple **SQL SELECT** statement to query data from the *products* table

Stored Procedures

Calling a Stored Procedure

To call a procedure, you only need to enter the word **CALL**, followed by the name of the procedure, and then the parentheses, including all the parameters between them (variables or values). Parentheses are compulsory (required).

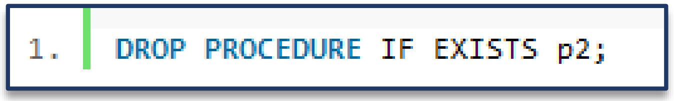


```
CALL stored_procedure_name (param1, param2, ....)

CALL procedure1(10 , 'string parameter' , @parameter_var);
```

Modify a Stored Procedure

MySQL provides an **ALTER PROCEDURE** statement to modify a routine, but only allows for the ability to change certain characteristics. If you need to alter the body or the parameters, you must **drop** and recreate the procedure

Drop (Delete) A Stored Procedure



```
DROP PROCEDURE IF EXISTS p2;
```

This is a simple command. The **IF EXISTS** clause prevents an error in case the procedure does not exist.

Listing

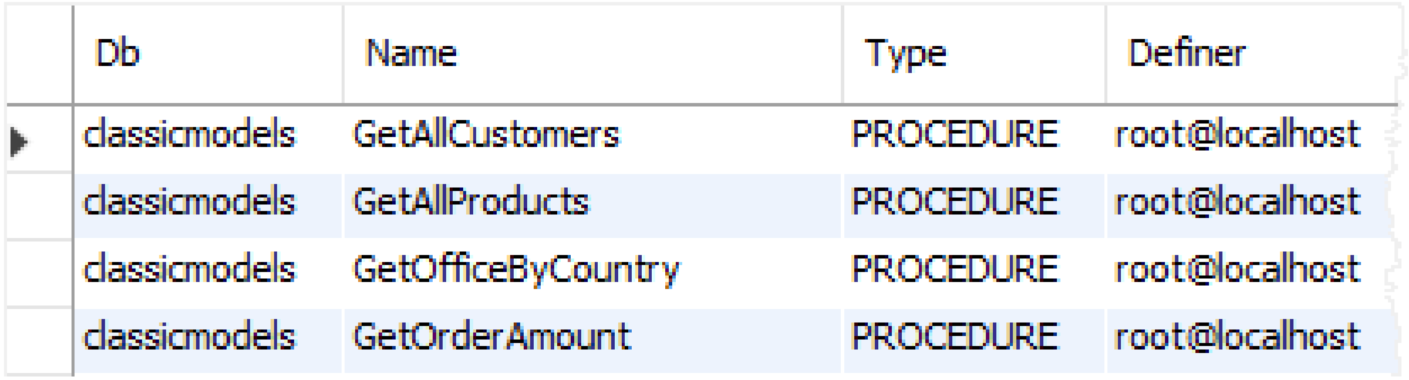
Here is the basic syntax of the `SHOW PROCEDURE STATUS` statement:

```
SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE search_condition]
```

The `SHOW PROCEDURE STATUS` statement shows all characteristic of stored procedures including stored procedure names. It returns stored procedures that you have a privilege to access.

The following statement shows all stored procedure in the current MySQL server:

```
SHOW PROCEDURE STATUS;
```



Db	Name	Type	Definer
classicmodels	GetAllCustomers	PROCEDURE	root@localhost
classicmodels	GetAllProducts	PROCEDURE	root@localhost
classicmodels	GetOfficeByCountry	PROCEDURE	root@localhost
classicmodels	GetOrderAmount	PROCEDURE	root@localhost

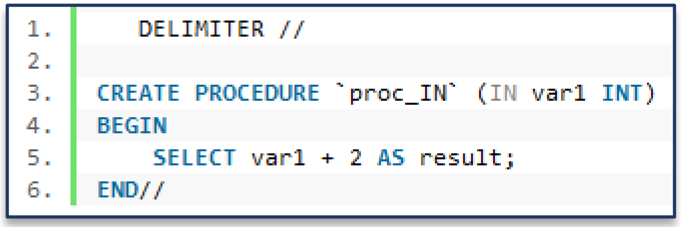
Parameters (in a Stored Procedure)

Let’s examine how you can define parameters within a stored procedure.

- **CREATE PROCEDURE proc1 ( )** : Parameter list is empty
- **CREATE PROCEDURE proc1 (IN *varname* DATA-TYPE)** : One input parameter. The word IN is optional because parameters are IN (input) by default.
- **CREATE PROCEDURE proc1 (OUT *varname* DATA-TYPE)** : One output parameter.
- **CREATE PROCEDURE proc1 (INOUT *varname* DATA-TYPE)** : One parameter which is both input and output.

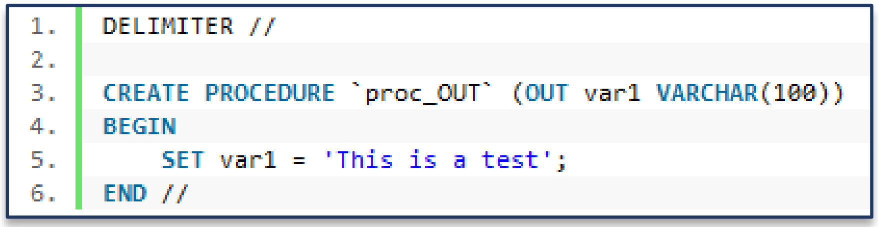
Of course, you can define multiple parameters defined with different types.

IN Example



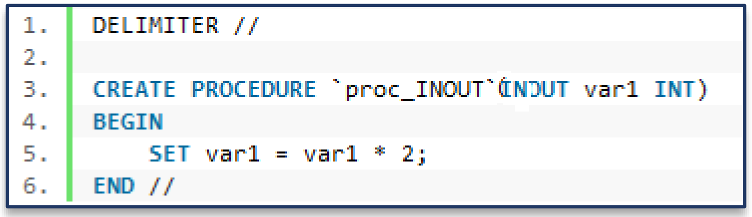
```
    DELIMITER //

CREATE PROCEDURE `proc_IN` (IN var1 INT)
BEGIN
    SELECT var1 + 2 AS result;
END//
```

OUT Example



```
DELIMITER //

CREATE PROCEDURE `proc_OUT` (OUT var1 VARCHAR(100))
BEGIN
    SET var1 = 'This is a test';
END //
```

INOUT Example

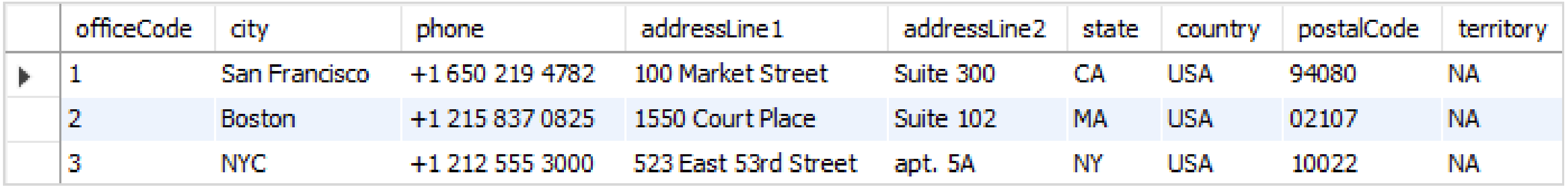


```
DELIMITER //

CREATE PROCEDURE `proc_INOUT`(INOUT var1 INT)
BEGIN
    SET var1 = var1 * 2;
END //
```

Parameters (IN)

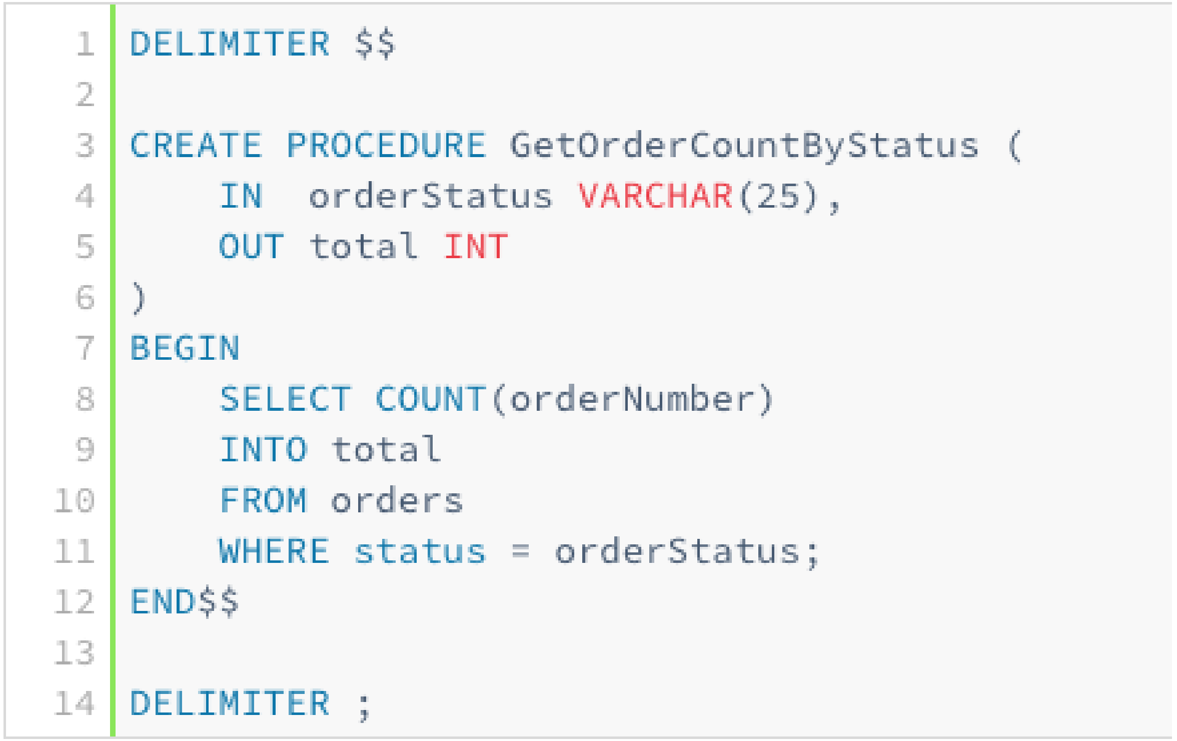
```sql
DELIMITER //

CREATE PROCEDURE GetOfficeByCountry(
    IN countryName VARCHAR(255)
)
BEGIN
    SELECT * 
     FROM offices
    WHERE country = countryName;
END //

DELIMITER ;
```

```sql
CALL GetOfficeByCountry('USA');
```

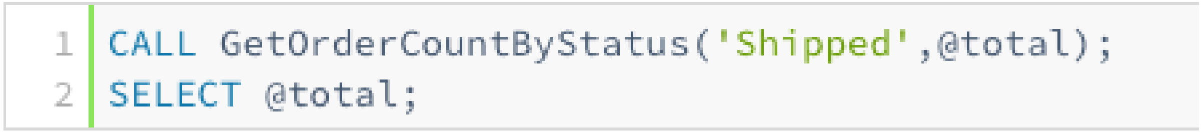


officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	NA
3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	NA

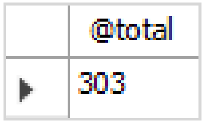
Parameters (OUT)



```sql
DELIMITER $$

CREATE PROCEDURE GetOrderCountByStatus (
    IN  orderStatus VARCHAR(25),
    OUT total INT
)
BEGIN
    SELECT COUNT(orderNumber)
    INTO total
    FROM orders
    WHERE status = orderStatus;
END$$

DELIMITER ;
```



```sql
CALL GetOrderCountByStatus('Shipped',@total);
SELECT @total;
```

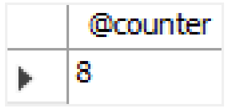


```
@total
303
```

Parameters (INOUT)

```sql
DELIMITER $$

CREATE PROCEDURE SetCounter(
    INOUT counter INT,
    IN inc INT
)
BEGIN
    SET counter = counter + inc;
END$$

DELIMITER ;
```

```sql
SET @counter = 1;
CALL SetCounter(@counter,1); -- 2
CALL SetCounter(@counter,1); -- 3
CALL SetCounter(@counter,5); -- 8
SELECT @counter; -- 8
```

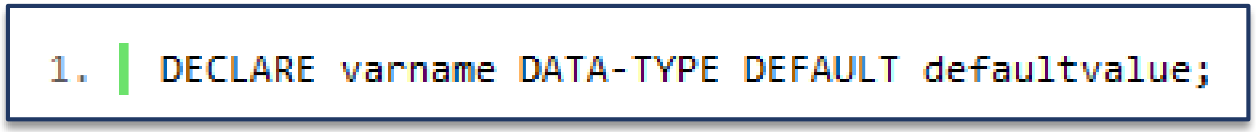


@counter
8

Variables (Stored Procedures)

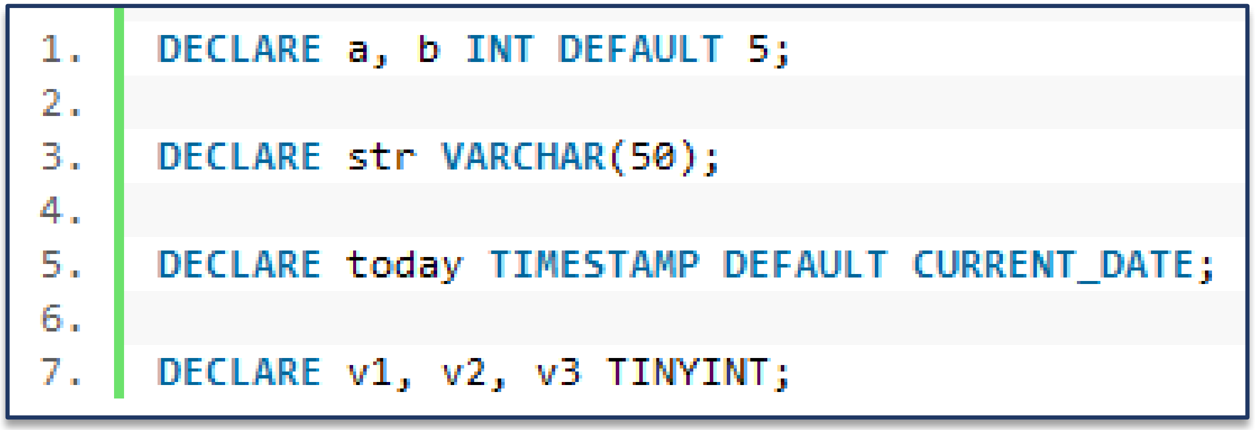
The following step will teach you how to define variables, and store values inside a procedure. You must declare them explicitly at the start of the BEGIN/END block, along with their data types. Once you've declared a variable, you can use it anywhere that you could use a session variable, or literal, or column name.

Declare a variable using the following syntax:



```
DECLARE varname DATA-TYPE DEFAULT defaultvalue;
```

Let's look at a few variables:

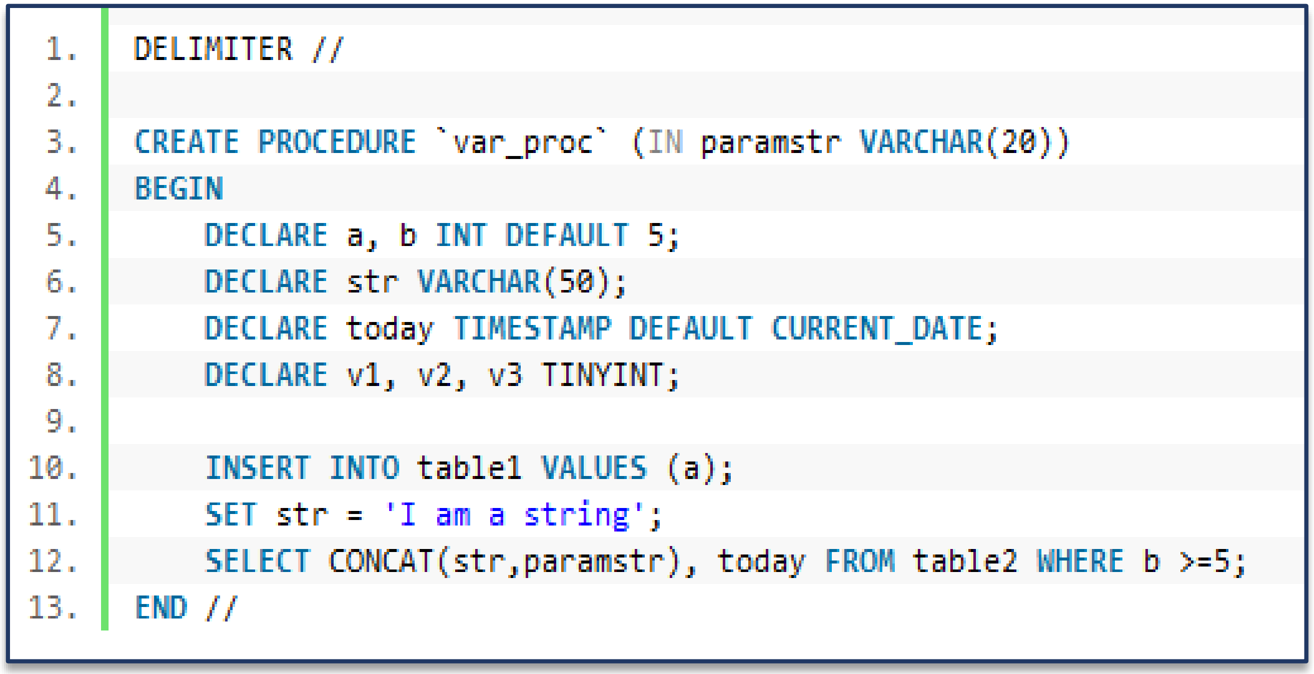


```
DECLARE a, b INT DEFAULT 5;

DECLARE str VARCHAR(50);

DECLARE today TIMESTAMP DEFAULT CURRENT_DATE;

DECLARE v1, v2, v3 TINYINT;
```

Variables (Stored Procedures)

Working with Variables

Once the variables have been declared, you can assign them values using the SET or SELECT command:

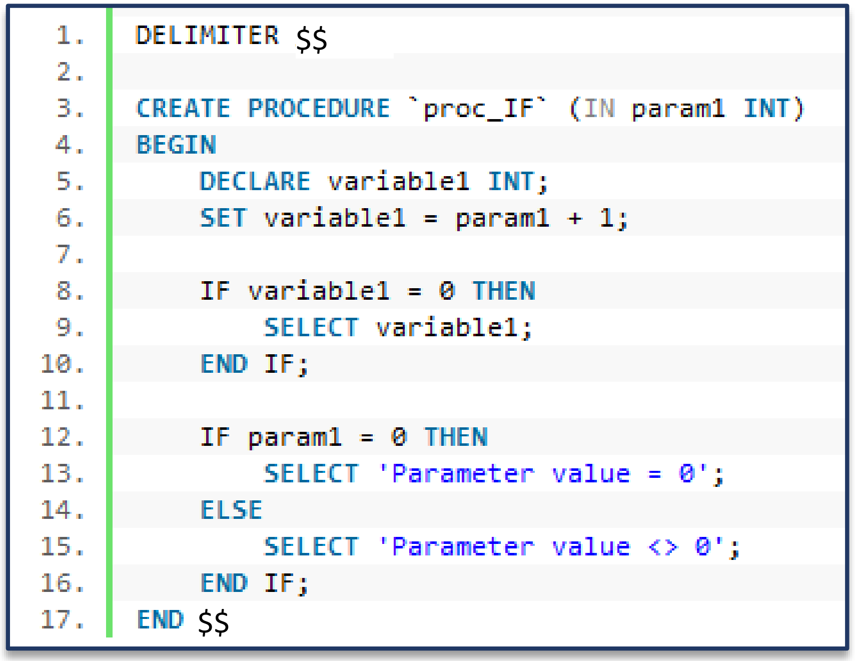


```sql
DELIMITER //

CREATE PROCEDURE `var_proc` (IN paramstr VARCHAR(20))
BEGIN
    DECLARE a, b INT DEFAULT 5;
    DECLARE str VARCHAR(50);
    DECLARE today TIMESTAMP DEFAULT CURRENT_DATE;
    DECLARE v1, v2, v3 TINYINT;

    INSERT INTO table1 VALUES (a);
    SET str = 'I am a string';
    SELECT CONCAT(str,paramstr), today FROM table2 WHERE b >=5;
END //
```

Flow Control Structures

MySQL supports the IF, CASE, ITERATE, LEAVE LOOP, WHILE and REPEAT constructs for flow control within stored programs. We're going to review how to use IF, CASE and WHILE specifically, since they happen to be the most commonly used statements in routines.

IF Statement

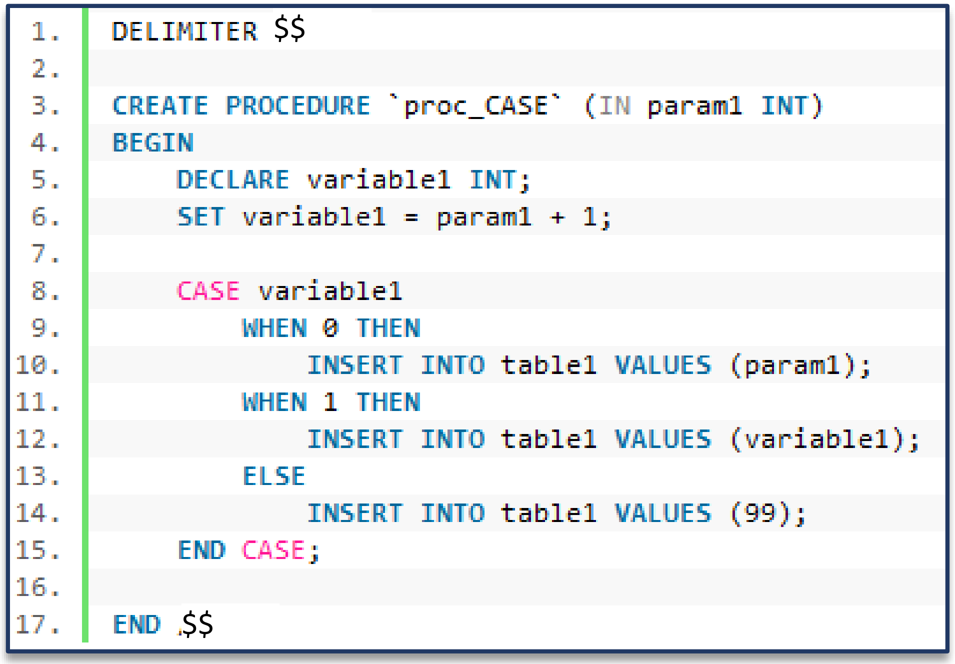


```
DELIMITER $$

CREATE PROCEDURE `proc_IF` (IN param1 INT)
BEGIN
    DECLARE variable1 INT;
    SET variable1 = param1 + 1;

    IF variable1 = 0 THEN
        SELECT variable1;
    END IF;

    IF param1 = 0 THEN
        SELECT 'Parameter value = 0';
    ELSE
        SELECT 'Parameter value <> 0';
    END IF;
END $$
```

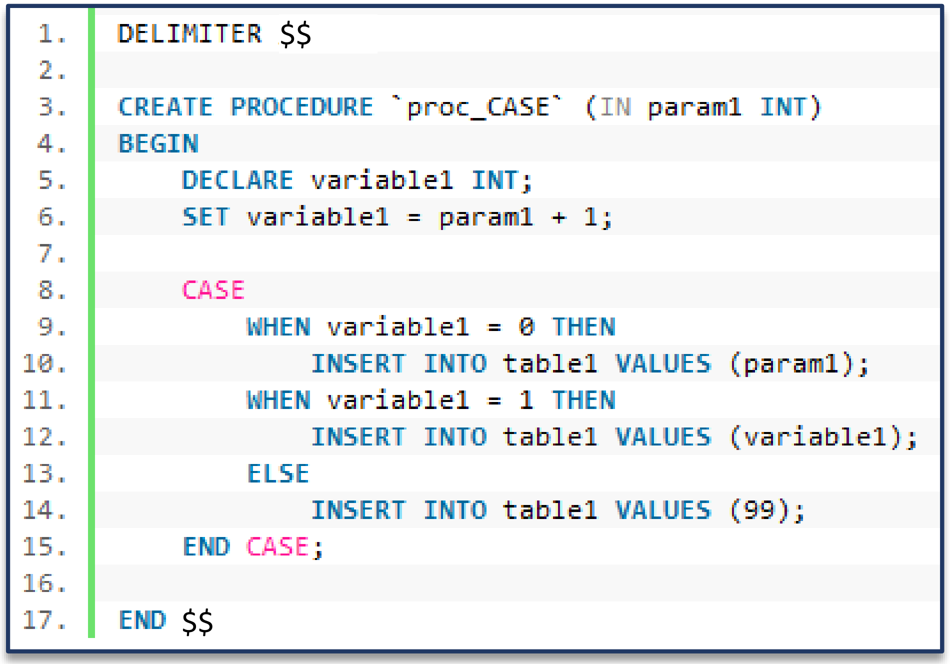
Flow Control Structures

CASE Statement

The CASE statement is another way to check conditions and take the appropriate path. It's an excellent way to replace multiple IF statements. The statement can be written in two different ways, providing great flexibility to handle multiple conditions



```sql
DELIMITER $$

CREATE PROCEDURE `proc_CASE` (IN param1 INT)
BEGIN
    DECLARE variable1 INT;
    SET variable1 = param1 + 1;

    CASE variable1
        WHEN 0 THEN
            INSERT INTO table1 VALUES (param1);
        WHEN 1 THEN
            INSERT INTO table1 VALUES (variable1);
        ELSE
            INSERT INTO table1 VALUES (99);
    END CASE;

END $$
```



```sql
DELIMITER $$

CREATE PROCEDURE `proc_CASE` (IN param1 INT)
BEGIN
    DECLARE variable1 INT;
    SET variable1 = param1 + 1;

    CASE
        WHEN variable1 = 0 THEN
            INSERT INTO table1 VALUES (param1);
        WHEN variable1 = 1 THEN
            INSERT INTO table1 VALUES (variable1);
        ELSE
            INSERT INTO table1 VALUES (99);
    END CASE;

END $$
```

Flow Control Structures

LOOP Statement

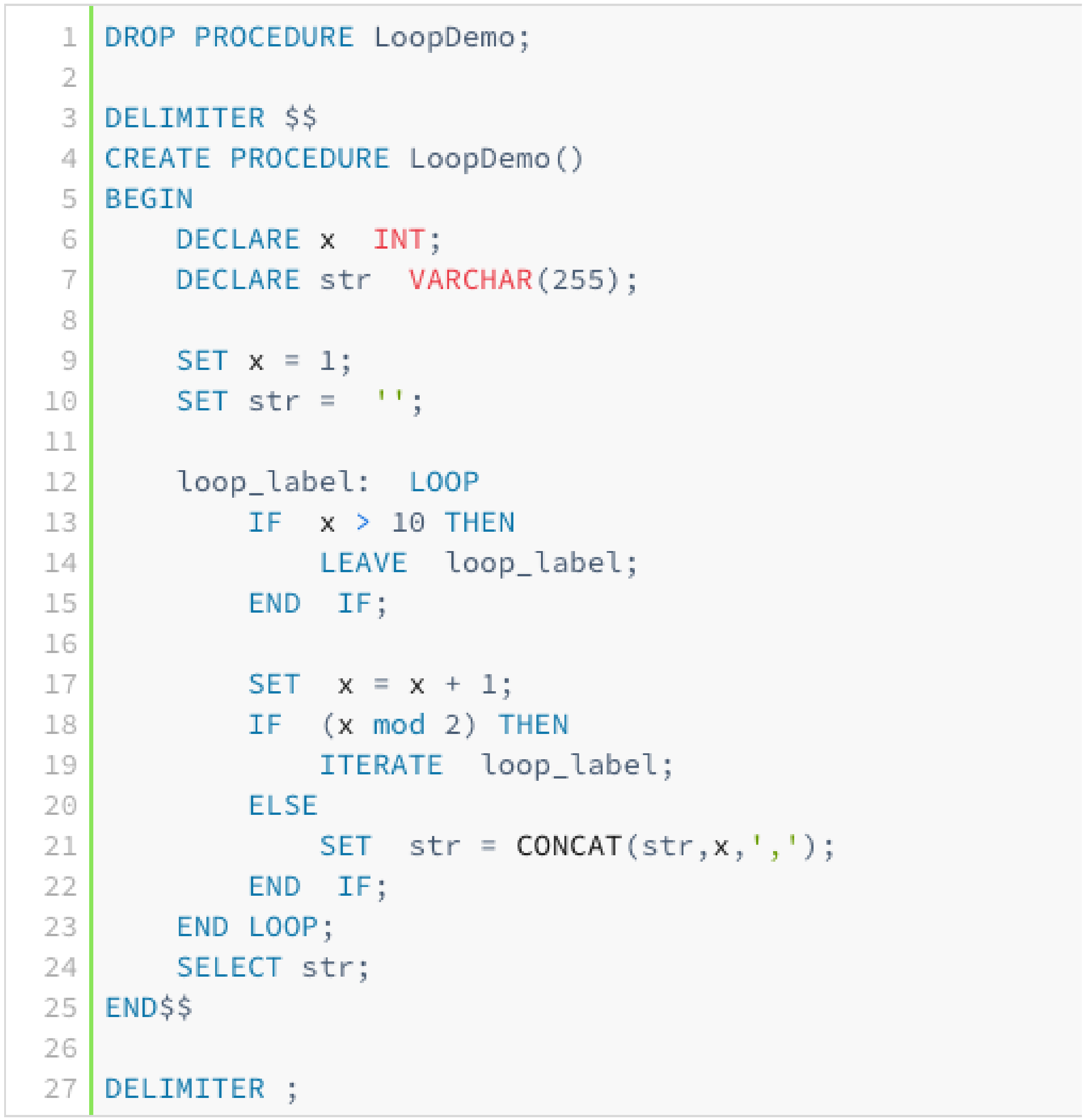
Syntax

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

Terminating LOOP

```
[label]: LOOP
    ...
    -- terminate the loop
    IF condition THEN
        LEAVE [label];
    END IF;
    ...
END LOOP;
```

Flow Control Structures (LOOP)

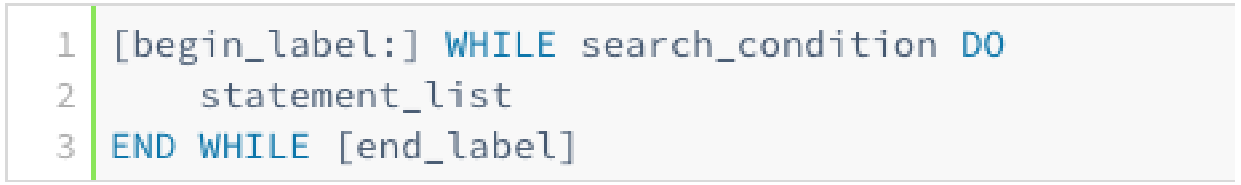


```
DROP PROCEDURE LoopDemo;

DELIMITER $$
CREATE PROCEDURE LoopDemo()
BEGIN
    DECLARE x  INT;
    DECLARE str  VARCHAR(255);

    SET x = 1;
    SET str =  '';

    loop_label:  LOOP
        IF  x > 10 THEN
            LEAVE  loop_label;
        END  IF;

        SET  x = x + 1;
        IF  (x mod 2) THEN
            ITERATE  loop_label;
        ELSE
            SET  str = CONCAT(str,x,',');
        END  IF;
    END LOOP;
    SELECT str;
END$$

DELIMITER ;
```

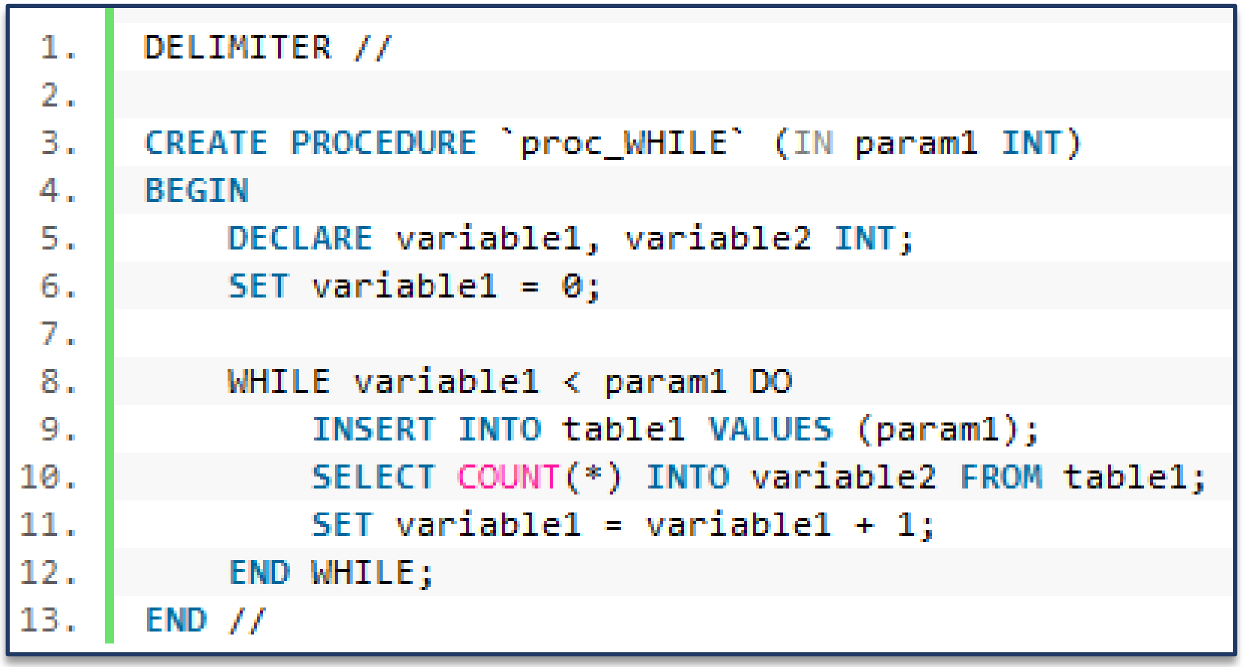
- The loop_label before the LOOPstatement for using with the ITERATE and LEAVE statements.
- ITERATE ignores everything below it and starts a new loop iteration
- LEAVE terminates the loop

Flow Control Structures

WHILE Statement



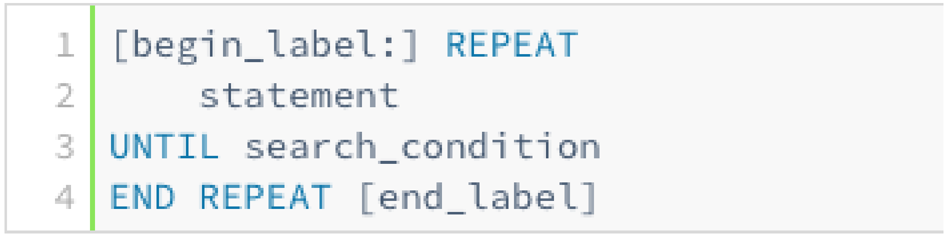
```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```



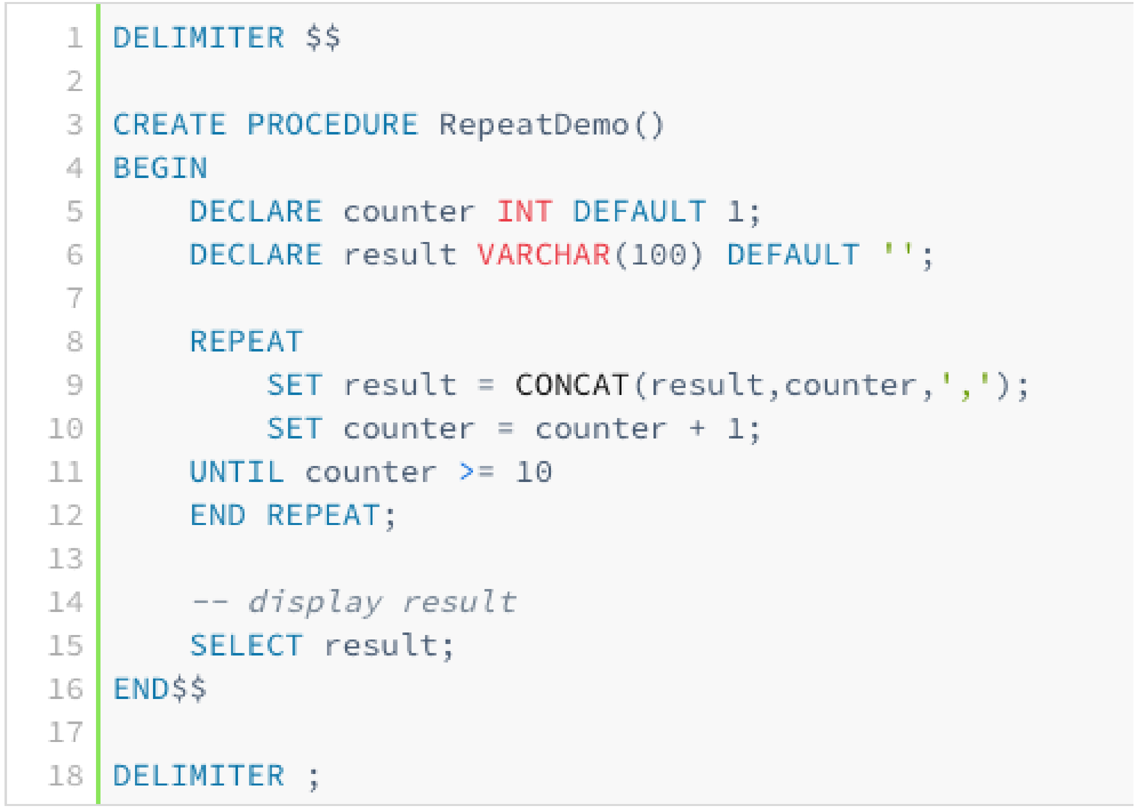
```
DELIMITER //

CREATE PROCEDURE `proc_WHILE` (IN param1 INT)
BEGIN
    DECLARE variable1, variable2 INT;
    SET variable1 = 0;

    WHILE variable1 < param1 DO
        INSERT INTO table1 VALUES (param1);
        SELECT COUNT(*) INTO variable2 FROM table1;
        SET variable1 = variable1 + 1;
    END WHILE;
END //
```

Flow Control Structures

REPEAT Statement



```
[begin_label:] REPEAT
    statement
UNTIL search_condition
END REPEAT [end_label]
```



```
DELIMITER $$

CREATE PROCEDURE RepeatDemo()
BEGIN
    DECLARE counter INT DEFAULT 1;
    DECLARE result VARCHAR(100) DEFAULT '';

    REPEAT
        SET result = CONCAT(result,counter,',');
        SET counter = counter + 1;
    UNTIL counter >= 10
    END REPEAT;

    -- display result
    SELECT result;
END$$

DELIMITER ;
```

Cursors (Stored Procedures)

"Cursor" is used to iterate through a set of rows returned by a query and process each row. MySQL supports cursor in stored procedures. Here's a summary of the essential syntax to create and use a cursor.

MySQL cursor is **read-only, non-scrollable and asensitive**

First, declare a cursor by using the DECLARE statement:

```
DECLARE cursor_name CURSOR FOR SELECT_statement;
```

The cursor declaration must be after any variable declaration. If you declare a cursor before the variable declarations, MySQL will issue an error. A cursor must always associate with a SELECT statement.
Next, open the cursor by using the OPEN statement. OPEN statement initializes the result set for the

```
OPEN cursor_name;
```

Cursors (Stored Procedures)

Then, use the FETCH statement to retrieve the next row pointed by the cursor and move the cursor to the next row in the result set.

```
FETCH cursor_name INTO variables list;
```

Finally, deactivate the cursor and release the memory associated with it using the CLOSE statement:

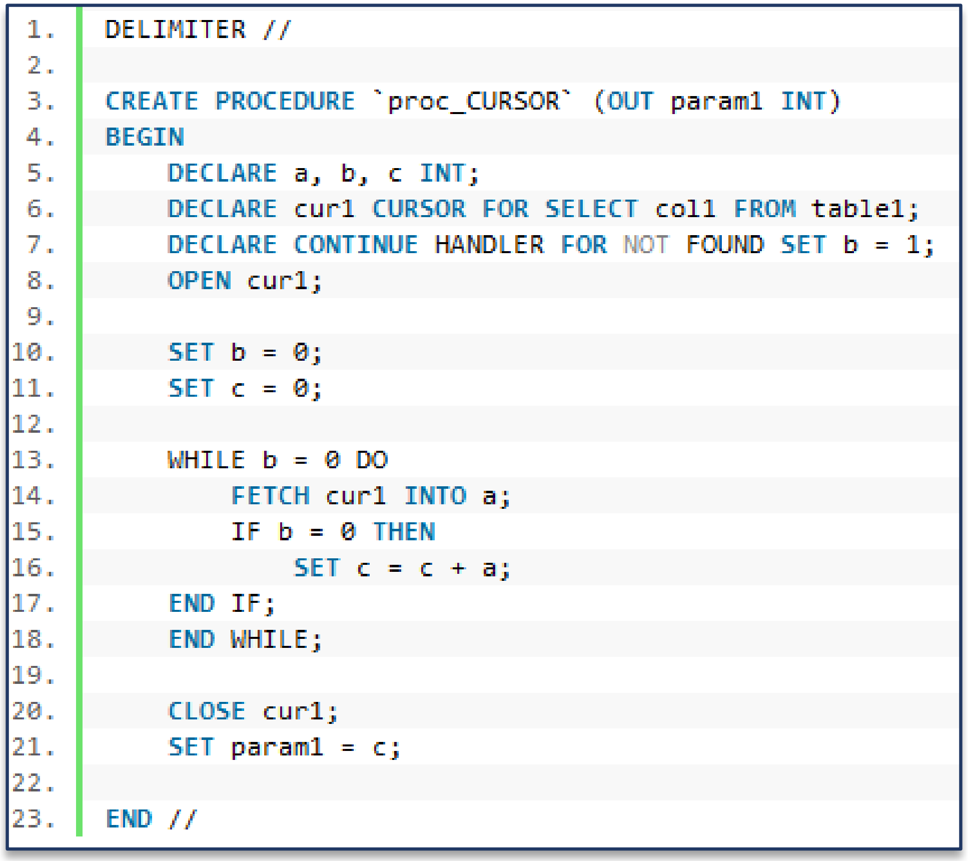
```
CLOSE cursor_name;
```

When working with MySQL cursor, you must also declare a NOT FOUND handler to handle the situation when the cursor could not find any row.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
```

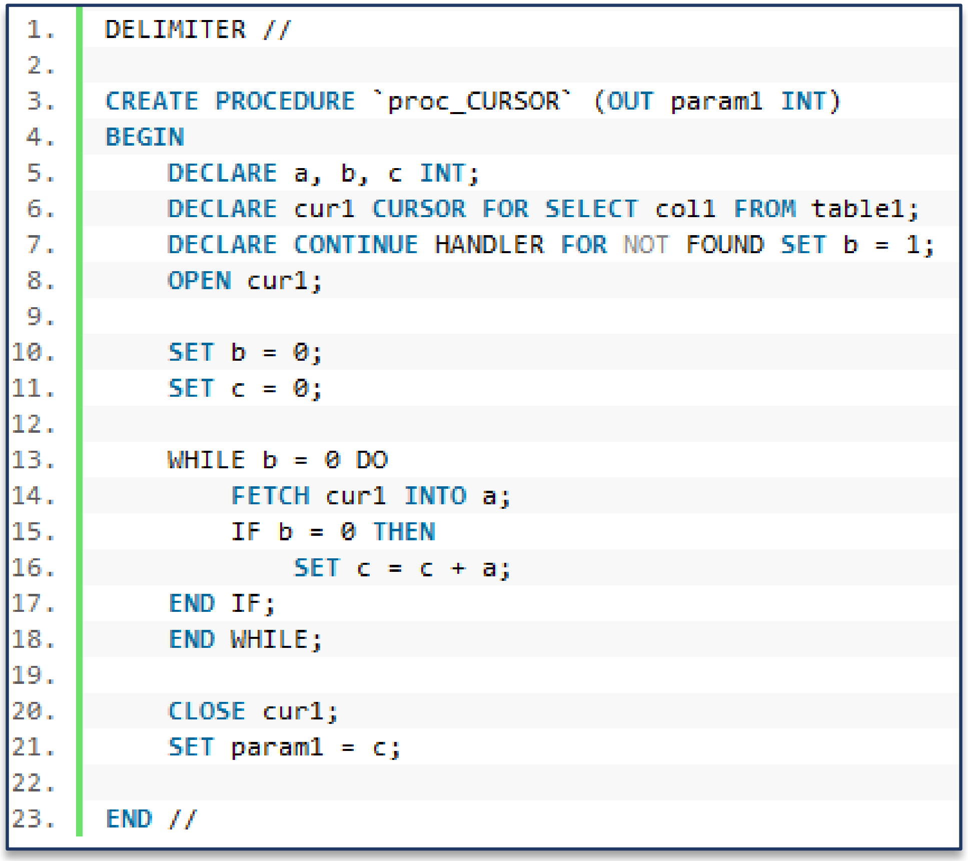
You can use MySQL cursors in stored procedures, stored functions, and triggers.

Cursors (Stored Procedures)

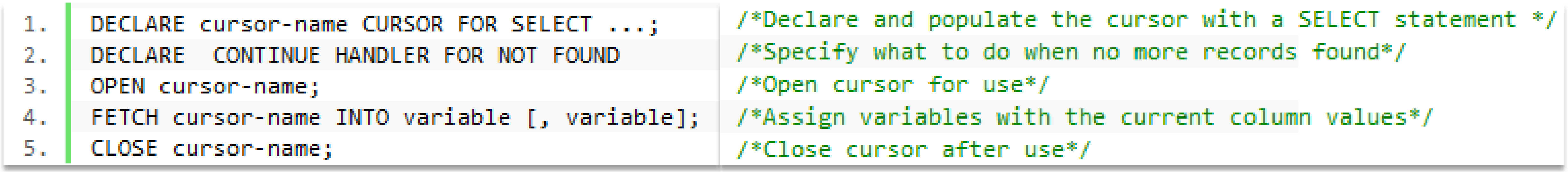


```
DELIMITER //

CREATE PROCEDURE `proc_CURSOR` (OUT param1 INT)
BEGIN
    DECLARE a, b, c INT;
    DECLARE cur1 CURSOR FOR SELECT col1 FROM table1;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = 1;
    OPEN cur1;

    SET b = 0;
    SET c = 0;

    WHILE b = 0 DO
        FETCH cur1 INTO a;
        IF b = 0 THEN
            SET c = c + a;
    END IF;
    END WHILE;

    CLOSE cur1;
    SET param1 = c;

END //
```

Cursors (Stored Procedures)



```
DELIMITER //

CREATE PROCEDURE `proc_CURSOR` (OUT param1 INT)
BEGIN
    DECLARE a, b, c INT;
    DECLARE cur1 CURSOR FOR SELECT col1 FROM table1;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = 1;
    OPEN cur1;

    SET b = 0;
    SET c = 0;

    WHILE b = 0 DO
        FETCH cur1 INTO a;
        IF b = 0 THEN
            SET c = c + a;
    END IF;
    END WHILE;

    CLOSE cur1;
    SET param1 = c;

END //
```



```
DECLARE cursor-name CURSOR FOR SELECT ...;        /*Declare and populate the cursor with a SELECT statement */
DECLARE  CONTINUE HANDLER FOR NOT FOUND           /*Specify what to do when no more records found*/
OPEN cursor-name;                                 /*Open cursor for use*/
FETCH cursor-name INTO variable [, variable];     /*Assign variables with the current column values*/
CLOSE cursor-name;                                /*Close cursor after use*/
```

HANDLER

```
DECLARE action HANDLER FOR condition_value statement;
```

If a condition whose value matches the condition_value , MySQL will execute the statement and continue or exit the current code block based on the action

The action accepts one of the following values:

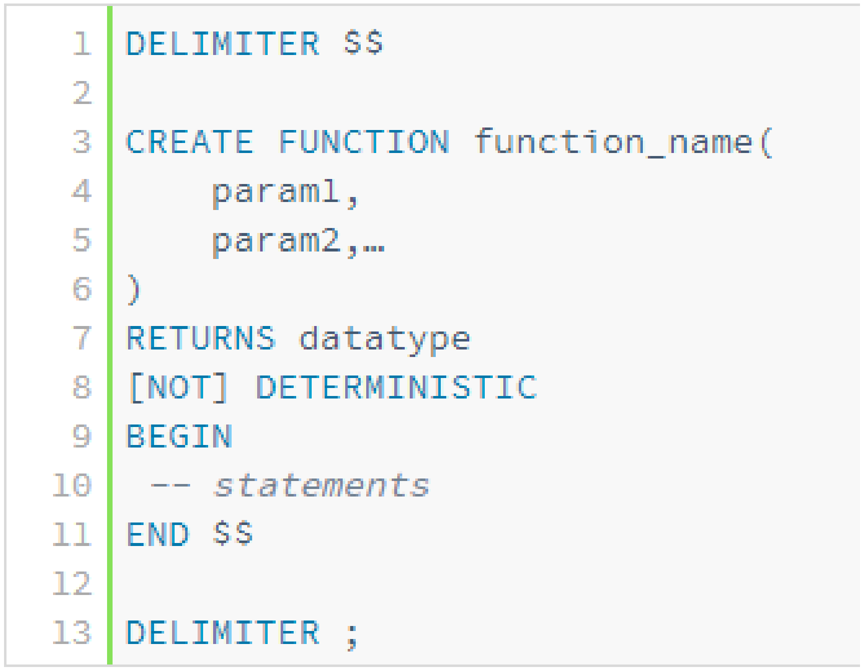
- CONTINUE : the execution of the enclosing code block continues.
- EXIT : the execution of the enclosing code block, where the handler is declared, terminates

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
SET hasError = 1;
```

Stored Function

A stored function is a special kind stored program that returns a **single value**. Typically, you use stored functions to encapsulate common formulas or business rules that are reusable among SQL statements or stored programs.

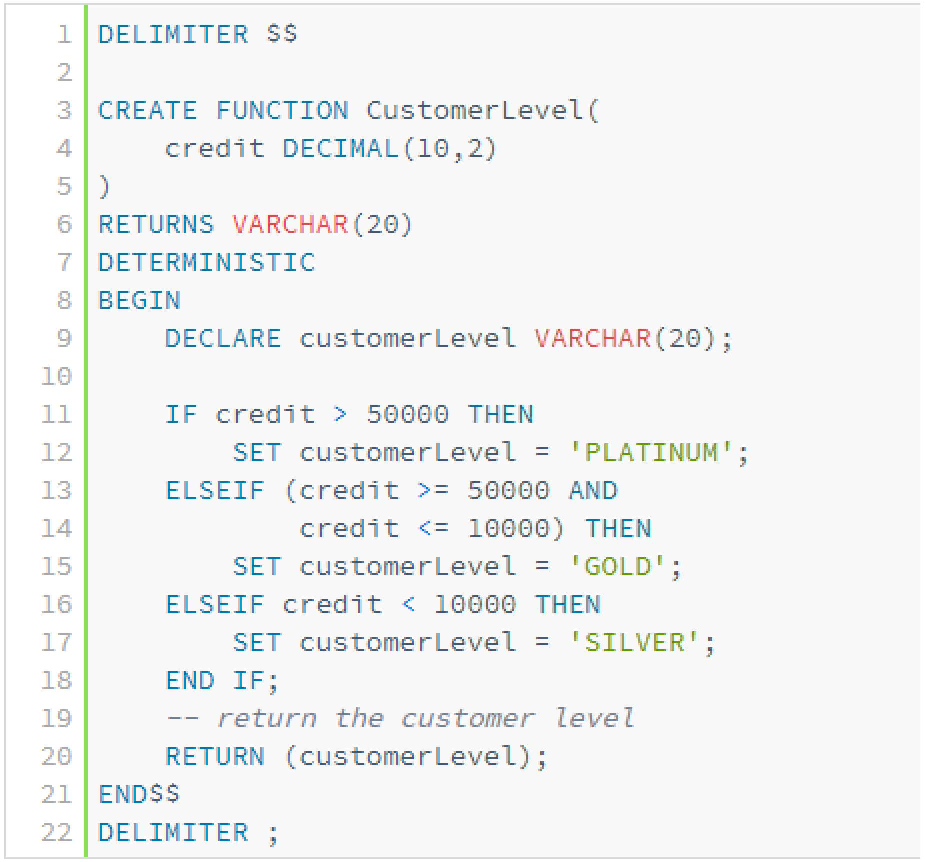
Different from a stored procedure, you can use a stored function in SQL statements wherever an expression is used.



```
DELIMITER $$

CREATE FUNCTION function_name(
    param1,
    param2,…
)
RETURNS datatype
[NOT] DETERMINISTIC
BEGIN
 -- statements
END $$

DELIMITER ;
```

By default, all parameters are the `IN` parameters

Stored Function



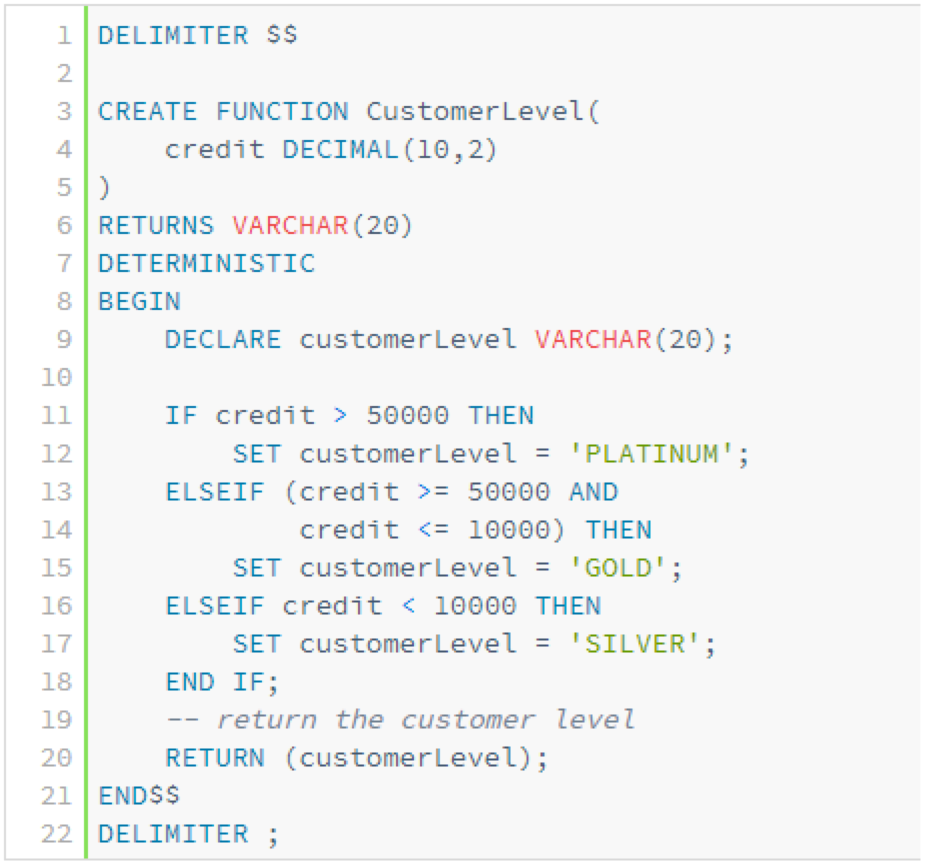
```sql
DELIMITER $$

CREATE FUNCTION CustomerLevel(
    credit DECIMAL(10,2)
)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE customerLevel VARCHAR(20);

    IF credit > 50000 THEN
        SET customerLevel = 'PLATINUM';
    ELSEIF (credit >= 50000 AND
            credit <= 10000) THEN
        SET customerLevel = 'GOLD';
    ELSEIF credit < 10000 THEN
        SET customerLevel = 'SILVER';
    END IF;
    -- return the customer level
    RETURN (customerLevel);
END$$
DELIMITER ;
```

Stored Function

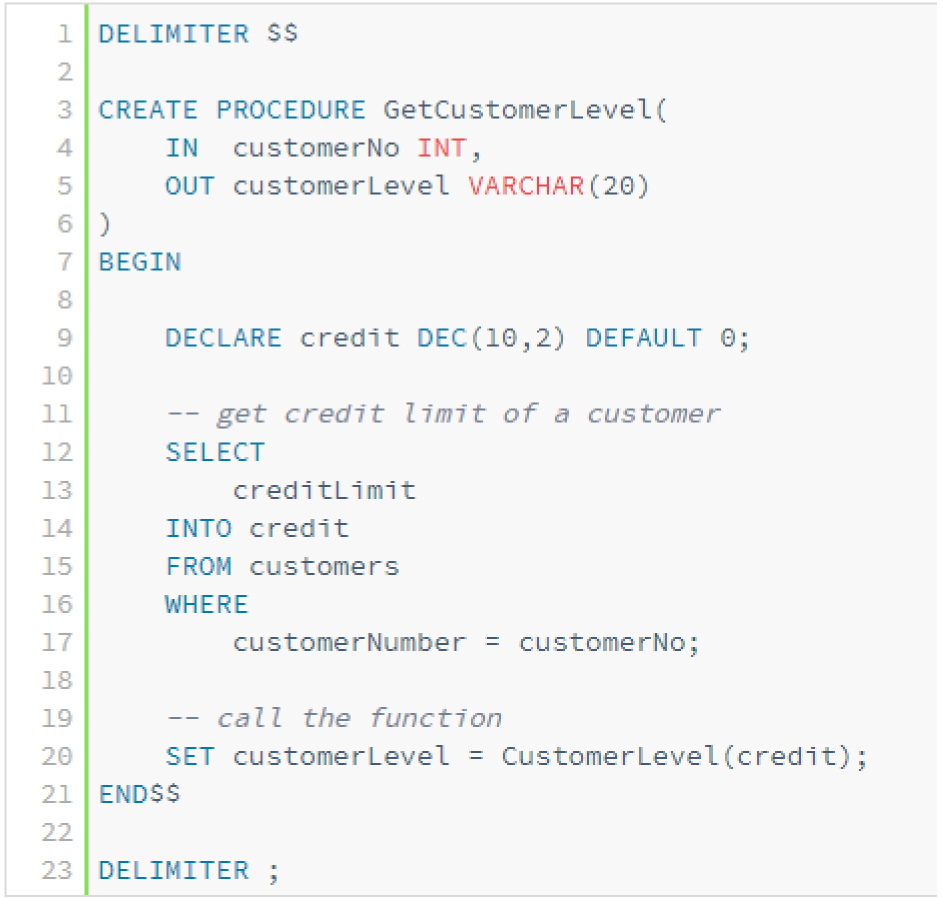
```
DELIMITER $$

CREATE FUNCTION CustomerLevel(
    credit DECIMAL(10,2)
)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE customerLevel VARCHAR(20);

    IF credit > 50000 THEN
        SET customerLevel = 'PLATINUM';
    ELSEIF (credit >= 50000 AND
            credit <= 10000) THEN
        SET customerLevel = 'GOLD';
    ELSEIF credit < 10000 THEN
        SET customerLevel = 'SILVER';
    END IF;
    -- return the customer level
    RETURN (customerLevel);
END$$
DELIMITER ;
```

```
SELECT
    customerName,
    CustomerLevel(creditLimit)
FROM
    customers
ORDER BY
    customerName;
```

Stored Function



```sql
DELIMITER $$

CREATE FUNCTION CustomerLevel(
    credit DECIMAL(10,2)
)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE customerLevel VARCHAR(20);

    IF credit > 50000 THEN
        SET customerLevel = 'PLATINUM';
    ELSEIF (credit >= 50000 AND
            credit <= 10000) THEN
        SET customerLevel = 'GOLD';
    ELSEIF credit < 10000 THEN
        SET customerLevel = 'SILVER';
    END IF;
    -- return the customer level
    RETURN (customerLevel);
END$$
DELIMITER ;
```



```sql
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
    IN  customerNo INT,
    OUT customerLevel VARCHAR(20)
)
BEGIN

    DECLARE credit DEC(10,2) DEFAULT 0;

    -- get credit limit of a customer
    SELECT
        creditLimit
    INTO credit
    FROM customers
    WHERE
        customerNumber = customerNo;

    -- call the function
    SET customerLevel = CustomerLevel(credit);
END$$

DELIMITER ;
```