Parameters

This lesson discusses the three different modes of parameters in a stored procedure.

We'll cover the following

Parameters

A parameter is a placeholder for a variable that is used to pass data to and from a stored procedure. An input parameter is used to pass data value to a stored procedure and an output parameter lets the stored procedure pass a data value back to the caller. A parameter can be defined by specifying the mode as well as the data type and an optional maximum length.

Parameters are used to make a stored procedure flexible. Another reason of using parameters is avoiding direct user inputs in a query string. A user input can result in a runtime error and in worst case a malicious input can potentially harm the system.

In MySQL a parameter can have three modes; **IN**, **OUT** and **INOUT**. If the mode of a parameter is defined as **IN**, it indicates that the application calling the stored procedure has to pass an argument. The stored procedure can not alter the value of the argument, rather it only works on a copy of the IN parameter and the original value of the parameter is retained after the stored procedure ends. A parameter defined as having **OUT** mode indicates that the stored procedure will pass an argument back to the caller. The value of an OUT parameter can change in the stored procedure and the new value is passed back in the end. The stored procedure, however, cannot access the initial value of the OUT parameter and its value is NULL when the procedure begins execution. The third mode, **INOUT** has properties of both the IN and

OUT mode. The caller may pass an argument to the stored procedure and the stored procedure can work on it and pass the altered value back to the caller.

Syntax

> [IN | OUT | INOUT] **parameter_name datatype** [(**length**)]

> Connect to the terminal below by clicking in the widget. Once connected, the command line prompt will show up. Enter or copy and paste the command **./DataJek/Lessons/53lesson.sh** and wait for the MySQL prompt to start-up.

```
-- The lesson queries are reproduced below for convenient copy/paste into the terminal.

-- Query 1
DELIMITER **
CREATE PROCEDURE GetActorsByNetWorth(IN NetWorth INT )
BEGIN
        SELECT * FROM Actors
        WHERE NetWorthInMillions >= NetWorth;
END **
DELIMITER ;

-- Query 2
CALL GetActorsByNetWorth(500);
CALL GetActorsByNetWorth(750);

-- Query 3
CALL GetActorsByNetWorth();

-- Query 4
DELIMITER **
CREATE PROCEDURE GetActorCountByNetWorth (
        IN  NetWorth INT,
        OUT ActorCount INT)
BEGIN
        SELECT COUNT(*) INTO ActorCount
        FROM Actors
        WHERE NetWorthInMillions >= NetWorth;
END**
DELIMITER ;

-- Query 5
CALL GetActorCountByNetWorth(500, @ActorCount);
SELECT @ActorCount;

-- Query 6
DELIMITER **
```

```
CREATE PROCEDURE IncreaseInNetWorth(
        INOUT Increase INT,
        IN ActorId INT )
BEGIN
        DECLARE OriginalNetWorth INT;

        SELECT NetWorthInMillions Into OriginalNetWorth
    FROM Actors
        WHERE Id = ActorId;

        SET Increase = OriginalNetWorth + Increase;
END**
DELIMITER ;

-- Query 7
SET @IncreasedWorth = 50;
CALL IncreaseInNetWorth(@IncreasedWorth, 11);
SELECT @IncreasedWorth;

-- Query 8
DELIMITER **
CREATE PROCEDURE GenderCountByNetWroth(
        IN NetWorth INT,
        OUT MaleCount INT,
        OUT FemaleCount INT)
BEGIN
        SELECT COUNT(*) INTO MaleCount
        FROM Actors
        WHERE NetWorthInMillions >= NetWorth
              AND Gender = 'Male';

        SELECT COUNT(*) INTO FemaleCount
        FROM Actors
        WHERE NetWorthInMillions >= NetWorth
              AND Gender = 'Female';
END**
DELIMITER ;

-- Query 9
CALL GenderCountByNetWroth(500, @Male, @Female);
SELECT @Male, @Female;
```

Terminal

1. Consider the example of a stored procedure that displays the list of actors whose net worth in millions is more than an input parameter **NetWorth**.

```
DELIMITER **

CREATE PROCEDURE GetActorsByNetWorth(
                   IN NetWorth INT )
BEGIN
     SELECT *
```

```
    FROM Actors
    WHERE NetWorthInMillions >= NetWorth;

END **

DELIMITER ;
```

Here we are using an input parameter **NetWorth** having an integer value and the stored procedure will return results based on this value. A stored procedure with an input parameter can be called as follows:

```
CALL GetActorsByNetWorth(500);

CALL GetActorsByNetWorth(750);
```

As it can be seen, the results of our stored procedure are now dependent on the input parameter. A call to this stored procedure without providing any input parameter will result in an error as shown below:

```
CALL GetActorsByNetWorth();
```

2. Now let's consider the example of a stored procedure that passes an argument back to the caller. We can modify the **GetActorsByNetWorth** procedure from step 1 to pass the number of actors whose **NetWorthInMillions** value is greater than the input parameter.

```
DELIMITER **

CREATE PROCEDURE GetActorCountByNetWorth (
    IN  NetWorth INT,
    OUT ActorCount INT
)
BEGIN
    SELECT COUNT(*)
    INTO ActorCount
    FROM Actors
    WHERE NetWorthInMillions >= NetWorth;
END**

DELIMITER ;
DELIMITER ;
```

We have specified two parameters for our stored procedure, one is input and the other output. To receive the return value, we will pass a session variable. A session variable is preceded by @ sign. The value returned by the stored procedure can then be displayed using a **SELECT** statement as follows:

```
CALL GetActorCountByNetWorth(500, @ActorCount);
SELECT @ActorCount;
```

3. When the mode of a parameter is **INOUT**, the stored procedure can access the original value of the parameter when it is passed to it and that becomes the parameter's initial value within the procedure. We will show this with the example of a stored procedure that takes a value to be added to the **NetWorthInMillions** value as follows:

```
DELIMITER **

CREATE PROCEDURE IncreaseInNetWorth(
    INOUT IncreasedWorth INT,
    IN ActorId INT,
    )
BEGIN
    DECLARE OriginalNetWorth INT;

    SELECT NetWorthInMillions Into OriginalNetWorth
FROM Actors WHERE Id = ActorId;

SET IncreasedWorth = OriginalNetWorth + IncreasedWorth;

END**
DELIMITER ;
```

To run the stored procedure execute the following:

```
SET @IncreasedWorth = 50;

CALL IncreaseInNetWorth(@IncreasedWorth, 11);

SELECT @IncreasedWorth
SELECT @increasedWorth;
```

This is a contrived example to show how the value of a parameter with **INOUT** mode changes. The value of **Increase** is 50 before the stored procedure is called. Within the procedure, the value is changed and the new value is displayed afterwards. When the mode is INOUT the stored procedure can take the initial value, modify it and return the new value. If we had passed **Increase** in IN mode then the stored procedure would not be allowed to modify its value.

4. A stored procedure can return multiple values back to the caller. This is in contrast to a stored function which can only return one value. We will discuss stored functions later. For now let's consider an example where a stored procedure returns the number of male and female actors whose net worth is more than an input amount.

```
DELIMITER **

CREATE PROCEDURE GenderCountByNetWroth(
    IN NetWorth INT,
    OUT MaleCount INT,
    OUT FemaleCount INT)
BEGIN
        SELECT COUNT(*) INTO MaleCount
        FROM Actors
        WHERE NetWorthInMillions >= NetWorth
              AND Gender = 'Male';

    SELECT COUNT(*) INTO FemaleCount
        FROM Actors
        WHERE NetWorthInMillions >= NetWorth
              AND Gender = 'Female';

END**
DELIMITER ;
```

In this stored procedure the two **OUT** parameters get their value based on the **IN** parameter **NetWorth**.

```
CALL GenderCountByNetWroth(500, @Male, @Female);
```

```sql
SELECT @Male, @Female;
```