

MySQL Server-Side Programming with

`classicmodels`

A teaching guide covering **transaction control**, **stored procedures**, **functions**, **triggers**, and **indexes**, using the `classicmodels` sample database (customers, orders, products, employees, offices, payments, orderdetails).

Each section has: a short explainer, one or more worked examples, and an exercise activity for students to complete on their own.

1. Transaction Control

A **transaction** groups several SQL statements into a single all-or-nothing unit of work: either every statement in it takes effect, or none of them do. This matters whenever related changes must stay consistent with each other — for example, moving stock between two tables, or recording an order and its order lines together.

1.1 START TRANSACTION , COMMIT , and ROLLBACK

```
START TRANSACTION;

UPDATE customers
SET creditLimit = creditLimit - 5000
WHERE customerNumber = 141;

UPDATE customers
SET creditLimit = creditLimit + 5000
WHERE customerNumber = 124;

COMMIT;
```

Explainer: `START TRANSACTION` marks the beginning of the unit of work. Every statement after it is provisional until `COMMIT` makes the changes permanent. If something goes wrong before you commit, `ROLLBACK` instead of `COMMIT` undoes everything back to the point the transaction started:

```
START TRANSACTION;

UPDATE customers
SET creditLimit = creditLimit - 5000
WHERE customerNumber = 141;

-- something has gone wrong – abandon both changes
ROLLBACK;
```

1.2 Using transactions to keep related inserts consistent

```
START TRANSACTION;

INSERT INTO orders (orderNumber, orderDate, requiredDate,
status, customerNumber)
VALUES (10500, CURDATE(), DATE_ADD(CURDATE(), INTERVAL 10
DAY), 'In Process', 141);

INSERT INTO orderdetails (orderNumber, productCode,
quantityOrdered, priceEach, orderLineNumber)
VALUES (10500, 'S18_1749', 5, 136.00, 1);

COMMIT;
```

Explainer: An order without any order lines (or order lines pointing at an order that doesn't exist) would leave the database in an inconsistent state. Wrapping both inserts in one transaction guarantees that either the whole order is recorded, or none of it is — there's no window where one exists without the other.

1.3 `SAVEPOINT` — partial rollback

```

START TRANSACTION;

UPDATE products SET quantityInStock = quantityInStock - 5
WHERE productCode = 'S18_1749';

SAVEPOINT after_stock_update;

UPDATE customers SET creditLimit = creditLimit - 500
WHERE customerNumber = 141;

-- decide the credit change shouldn't happen, but keep the
stock change
ROLLBACK TO after_stock_update;

COMMIT;

```

Explainer: A `SAVEPOINT` marks a point inside a transaction that you can roll back to without undoing everything. `ROLLBACK TO after_stock_update` undoes only the statements after the savepoint; the stock update made before it survives and is written permanently once you `COMMIT`.

1.4 Isolation and `autocommit`

```

SELECT @@autocommit;

SET autocommit = 0;
-- statements now require an explicit COMMIT or ROLLBACK
SET autocommit = 1;

```

Explainer: By default MySQL runs in `autocommit` mode, where every individual statement is treated as its own transaction and committed immediately. Turning `autocommit` off means changes only become permanent when you explicitly `COMMIT` — useful when a script needs to run several statements as one logical operation without wrapping every block in `START TRANSACTION`.

🧩 Exercise 1 — Transaction Control

Using the `classicmodels` database:

1. Write a transaction that transfers stock: decrease `quantityInStock` by 10 for one product and increase it by 10 for another, committing only if both updates succeed.
2. Write a transaction that inserts a new payment into `payments` and, in the same transaction, updates a status field of your choosing — then deliberately `ROLLBACK` instead of committing, and confirm with a `SELECT` that neither change took effect.
3. Repeat the stock transfer from Q1, but place a `SAVEPOINT` after the first update. Roll back to the savepoint and explain, in a comment, what state the data is left in and why.

2. Stored Procedures

A **stored procedure** is a named, saved block of SQL statements that lives inside the database and can be called by name. Procedures can take input/output parameters and can contain logic (`IF` , loops, variables) — they don't have to just return a single value.

2.1 Basic procedure — customer orders lookup

```
DELIMITER $$

CREATE PROCEDURE GetOrdersByCustomer (IN custNumber INT)
BEGIN
    SELECT orderNumber, orderDate, status
    FROM orders
    WHERE customerNumber = custNumber
    ORDER BY orderDate DESC;
END $$

DELIMITER ;
```

Call it with:

```
CALL GetOrdersByCustomer(141);
```

Explainer: DELIMITER \$\$ temporarily changes the statement terminator from ; to \$\$ so that the semicolons *inside* the procedure body don't end the CREATE PROCEDURE statement early. IN custNumber INT declares an input parameter. The procedure body sits between BEGIN and END .

2.2 Procedure with an OUT parameter and logic

```
DELIMITER $$

CREATE PROCEDURE CheckCreditStatus (
    IN custNumber INT,
    OUT statusMessage VARCHAR(100)
)
BEGIN
    DECLARE totalOwed DECIMAL(10,2);
    DECLARE limitAmt DECIMAL(10,2);

    SELECT creditLimit INTO limitAmt
    FROM customers
    WHERE customerNumber = custNumber;

    SELECT IFNULL(SUM(o.quantityOrdered * o.priceEach), 0)
    INTO totalOwed
    FROM orders ord
    JOIN orderdetails o ON ord.orderNumber = o.orderNumber
    WHERE ord.customerNumber = custNumber
        AND ord.status NOT IN ('Cancelled', 'Resolved');

    IF totalOwed > limitAmt THEN
        SET statusMessage = 'Over credit limit';
    ELSE
        SET statusMessage = 'Within credit limit';
    END IF;
END $$

DELIMITER ;
```

Call it with:

```
CALL CheckCreditStatus(141, @result);
SELECT @result;
```

Explainer: `DECLARE` creates local variables inside the procedure. `SELECT ... INTO variable` stores a query's single-row result into a variable instead of returning it as a result set. `OUT` parameters send a value back to the caller — you retrieve it afterwards with a session variable (`@result`).

🌱 Exercise 2 — Stored Procedures

1. Write a stored procedure `GetProductsByLine(IN line VARCHAR(50))` that returns `productCode`, `productName`, and `MSRP` for every product in a given product line.
2. Write a stored procedure `CountCustomerOrders(IN custNumber INT, OUT orderCount INT)` that counts how many orders a customer has placed, and stores the result in `orderCount`.
3. Test both procedures using at least two different customer numbers / product lines from the data.

3. Functions

A **stored function** is similar to a procedure, but it must return exactly one value and can be used directly inside a `SELECT` statement, just like a built-in function such as `NOW()` or `SUM()`.

3.1 Basic function — order total value

```
DELIMITER $$

CREATE FUNCTION OrderTotal(orderNum INT)
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    DECLARE total DECIMAL(10,2);

    SELECT SUM(quantityOrdered * priceEach) INTO total
```

```

FROM orderdetails
WHERE orderNumber = orderNum;

RETURN IFNULL(total, 0);
END $$

DELIMITER ;

```

Use it like a normal function:

```

SELECT orderNumber, OrderTotal(orderNumber) AS totalValue
FROM orders
WHERE customerNumber = 141;

```

Explainer: RETURNS DECIMAL(10,2) declares the data type of the single value the function gives back. DETERMINISTIC tells MySQL the function always returns the same output for the same input (important for replication and query optimisation — omit it, or use NOT DETERMINISTIC, if the function relies on something that changes, like NOW()). RETURN ends execution and hands back the value.

3.2 Function with conditional logic — customer tier

```

DELIMITER $$

CREATE FUNCTION CustomerTier(custNumber INT)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE limitAmt DECIMAL(10,2);

    SELECT creditLimit INTO limitAmt
    FROM customers
    WHERE customerNumber = custNumber;

    RETURN CASE
        WHEN limitAmt >= 100000 THEN 'Platinum'
        WHEN limitAmt >= 50000 THEN 'Gold'
        WHEN limitAmt > 0 THEN 'Standard'
        ELSE 'No Credit'
    END;
END;

```

```
END $$
```

```
DELIMITER ;
```

```
SELECT customerName, creditLimit,  
CustomerTier(customerNumber) AS tier  
FROM customers  
ORDER BY creditLimit DESC;
```

Exercise 3 — Functions

1. Write a function `ProfitMargin(productCode VARCHAR(15))` that returns the percentage profit margin for a product, calculated as $(\text{MSRP} - \text{buyPrice}) / \text{MSRP} * 100$, rounded to 2 decimal places.
2. Write a function `EmployeeFullName(empNumber INT)` that returns an employee's first and last name concatenated as a single string, e.g. 'Diane Murphy'.
3. Use both functions inside a single `SELECT` query that lists products with their profit margins, and a separate query that lists employees with their full names and job titles.

4. Triggers

A **trigger** is a block of SQL that runs automatically in response to an `INSERT`, `UPDATE`, or `DELETE` on a table — you never call it directly.

4.1 BEFORE INSERT trigger — data validation

```
DELIMITER $$  
  
CREATE TRIGGER trg_CheckCreditLimit  
BEFORE INSERT ON customers  
FOR EACH ROW  
BEGIN  
    IF NEW.creditLimit < 0 THEN  
        SET NEW.creditLimit = 0;  
    END IF;  
END IF;
```

```
END $$

DELIMITER ;
```

Explainer: `NEW` refers to the row about to be inserted (or updated). Because this is a `BEFORE` trigger, changing `NEW.creditLimit` here actually changes the value that gets stored — this is a common pattern for cleaning or validating incoming data.

4.2 `AFTER INSERT` trigger — audit logging

```
CREATE TABLE order_audit (
    auditId      INT AUTO_INCREMENT PRIMARY KEY,
    orderNumber  INT,
    action       VARCHAR(20),
    actionDate   DATETIME
);

DELIMITER $$

CREATE TRIGGER trg_LogNewOrder
AFTER INSERT ON orders
FOR EACH ROW
BEGIN
    INSERT INTO order_audit (orderNumber, action,
        actionDate)
        VALUES (NEW.orderNumber, 'INSERTED', NOW());
END $$

DELIMITER ;
```

Explainer: `AFTER` triggers run once the original operation has already completed successfully, so they're commonly used for logging, auditing, or cascading changes to other tables — here, every new order automatically writes a row to an audit table.

4.3 `BEFORE UPDATE` trigger — protecting shipped orders

```
DELIMITER $$
```

```

CREATE TRIGGER trg_PreventShippedEdit
BEFORE UPDATE ON orders
FOR EACH ROW
BEGIN
    IF OLD.status = 'Shipped' AND NEW.status != 'Shipped'
    THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot change the status of an
order that has already shipped.';
    END IF;
END $$

DELIMITER ;

```

Explainer: `OLD` refers to the row's values before the update; `NEW` refers to the incoming values. `SIGNAL SQLSTATE '45000'` raises a custom error, which cancels the update and returns the message text to whatever application or client issued the `UPDATE` .

🌱 Exercise 4 — Triggers

1. Create a `product_audit` table, then write an `AFTER UPDATE` trigger on `products` that logs the `productCode` , the old `MSRP` , the new `MSRP` , and the date whenever a product's `MSRP` changes.
2. Write a `BEFORE INSERT` trigger on `payments` that rejects (via `SIGNAL`) any payment with an `amount` of zero or less.
3. Test your triggers by running `UPDATE` / `INSERT` statements that should succeed, and ones that should be blocked or logged, and confirm the behaviour matches.

5. Indexes

An **index** is a data structure MySQL builds alongside a table to make lookups on particular columns much faster, at the cost of extra storage and slightly slower writes.

5.1 Single-column index

```
CREATE INDEX idx_customers_country
ON customers (country);
```

Explainer: Speeds up queries that filter or sort by `country`, e.g. `WHERE country = 'UK'`. Without this index, MySQL must scan every row in the table (a "full table scan") to find matches.

5.2 Composite (multi-column) index

```
CREATE INDEX idx_orderdetails_order_product
ON orderdetails (orderNumber, productCode);
```

Explainer: A composite index helps queries that filter on the leading column (`orderNumber`) alone, or on both columns together. Column order matters: this index helps `WHERE orderNumber = 10100`, but is far less useful for `WHERE productCode = 'S18_1749'` on its own.

5.3 Unique index

```
CREATE UNIQUE INDEX idx_employees_email
ON employees (email);
```

Explainer: A unique index both speeds up lookups and enforces that no two rows can share the same value — MySQL will reject an `INSERT` or `UPDATE` that would create a duplicate email address.

5.4 Checking whether an index is being used

```
EXPLAIN SELECT customerName
FROM customers
WHERE country = 'UK';
```

Explainer: `EXPLAIN` shows the query plan MySQL intends to use — look at the `key` column to see whether it picked up `idx_customers_country`, and `rows` to see roughly how many rows it expects to examine.

🌱 Exercise 5 — Indexes

1. Run `EXPLAIN` on a query that filters `orders` by `status = 'Disputed'` — note the `rows` and `type` values.
2. Create an index on `orders(status)`, then re-run the same `EXPLAIN` and compare the results.
3. Create a composite index on `payments(customerNumber, paymentDate)` and explain, in a sentence or two, which queries would benefit from it and why column order matters here.
4. Drop the index you created in Q2 using `DROP INDEX idx_name ON orders;` once you've finished comparing.

Summary Table

Feature	Returns / Effect	Called by	Typical use
Transaction control	Commits or undoes a group of statements	<code>START TRANSACTION / COMMIT / ROLLBACK</code>	Keeping related changes consistent
Stored procedure	Zero or more result sets, <code>OUT</code> params	<code>CALL procedure_name(...)</code>	Reusable multi-step logic
Function	Exactly one value	Inside a <code>SELECT</code> /expression	Reusable calculations
Trigger	N/A — runs automatically	Fired by <code>INSERT / UPDATE / DELETE</code>	Validation, auditing, enforcement

Feature	Returns / Effect	Called by	Typical use
Index	N/A — speeds up lookups	Used automatically by the optimiser	Query performance