Before we get started with **SELECT statements**, we'll look at some basic **SQL syntax** rules that apply to all
SQL statements
:

- Comments
- Whitespace
- Semi-colon use
- Case Sensitivity

Comments in SQL

The standard SQL comment is two hyphens (--). However, some databases use other forms of comments as shown in the table below.

SQL Comments

	--	#	/* */
Example	-- Comment	# Comment	/* Comment */
ANSI	YES	NO	NO
SQL Server	YES	NO	YES
Oracle	YES	NO	YES
MySQL	YES	YES	YES

The code sample below shows some sample comments.

Code Sample:

```
     -- Single-line comment
/*
 Multi-line comment used in:
   -SQL Server
   -Oracle
   -MySQL
```

*/

Whitespace and Semi-colons

Whitespace is ignored in SQL statements. Multiple statements are separated with semi-colons. The two statements in the sample below are equally valid.

Code Sample:

```
      SELECT * FROM category;

SELECT
last_name,

first_name, email
FROM staff;
```

Case Sensitivity

SQL is basically not case sensitive. It is common practice to write reserved words in all capital letters. User-defined names, such as table names and column names may or may not be case sensitive depending on the operating system used. The case-sensitivity aspect has been covered in another lesson.

SELECT Basics

Whenever you want to retrieve data from a **MySQL database**, you can issue a **SELECT statement**
that specifies what data you want to have returned and in what manner that data should be returned. For example, you can specify that only specific columns or rows be returned. You can also order the rows based on the values in one or more columns. In addition, you can group together rows based on repeated values in a column in order to summarize data.

Referring back to the syntax, notice that a **SELECT syntax** requires only the following clause:

```
    SELECT {* | <select list>}
```

The SELECT clause includes the **SELECT keyword** and an asterisk (*) or the select list, which is made up of columns or expressions, as shown in the following syntax:

```
    Syntax  <select> [ * | <column name> | <expression>} [[AS] <alias>]
[{, {<column name> | <expression>} [[AS] <alias>]}...]
```

```
FROM <table name> [[AS] <alias>]
[WHERE <expression> [{<operator> <expression>}...]]
[ORDER BY <column name> [ASC | DESC] [{, <column name> [ASC | DESC]}...]]
[LIMIT [<offset>,] <row count>]
```

As you can see, the select list must include at least one column name or one expression. If more than one column/expression element is included, they must be separated by commas. In addition, you can assign an alias to a column name or expression by using the AS subclause. That alias can be then be used in other clauses in the
SELECT
statement
; however, it cannot be used in a
WHERE
clause
because of the way in which MySQL processes a
SELECT
statement
.

Note: As the syntax indicates, the AS keyword is optional when assigning an alias to a column. For the sake of clarity and to avoid confusion with other column names, it is generally recommended that you include the AS keyword.

Although the **SELECT clause** is the only required element in a **SELECT statement**, you cannot retrieve data from a table unless you also specify a
FROM
clause and the appropriate table.

Introduction to the *sakila* Database

The *sakila* database is a sample database used by **MySQL** to demonstrate the various statements used in MySQL. The database contains the rental data on films stored in
sakila
database. The
sakila sample database
is available from
http://dev.mysql.com/doc/
. A downloadable archive is available in compressed tar file or Zip format.

Note: We use the *sakila* database for many of our examples as there are many resources on MySQL website for related learning using this database.

A document explaining and showing the ER diagram of the table structure is available for *sakila* database on MySQL website.

The *sakila* database has a good set of well-connected tables, and we will create some additional tables of our own modeled after these tables to demonstrate some key ideas.

The *SetupDirectory* contains the original sakila schema and data scripts, and course-specific scripts. The techniques and syntax used in these scripts are beyond the scope of this lesson, but will become clearer as we progress through other lessons:

Here are the scripts to run initially:

1. sakila-schema.sql
2. sakila-data.sql
3. sakila-own-schema.sql

Basic Select Examples

SELECTing All Columns in All Rows

The following query is used to retrieve all columns in all rows of a table.

Syntax
```
SELECT table_name.* FROM table;
-- OR
SELECT * FROM table;
```

Code Sample:

```
--Retrieve all columns from the customer table
SELECT *
FROM customer;
```

SELECTing Specific Columns

The following query is used to retrieve specific columns in all rows of a table.

```
    Syntax  SELECT table_name.column_name, table_name.column_name
FROM table;

-- OR

SELECT column, column
FROM table;
```

Code Sample:

```
      /*
Select the film_title and rating from the film table.
*/
SELECT title, rating
FROM film;
```

Code Explanation

Select the title and rating from the film table.

The WHERE Clause and Basic Operator Symbols

The **WHERE clause** is used to retrieve specific rows from tables. The **WHERE clause** can contain one or more conditions that specify which rows should be returned.

```
    Syntax  SELECT column, column
FROM table
WHERE conditions;
```

In this section, you look at the **WHERE clause**, which allows you to specify which rows in your table are returned by your query.

The **WHERE clause** is made up of one or more conditions that define the parameters of the **SE LECT**
statement
. Each condition is an expression that can consist of column names, literal values, operators, and functions. The following syntax describes how a
WHERE
clause is defined:

```
    WHERE <expression> [{<operator> <expression>}...]
```

As you can see, a **WHERE clause** must contain at least one expression that defines which rows the **SELECT statement** returns. When you specify more than one condition in the
WHERE
clause, those conditions are connected by an
AND
or an
OR
operator. The operators specify which condition or combination of conditions must be met.

The less than (<) and greater than (>) signs are used to compare numbers, dates, and strings.

Note that non-numeric values (e.g, dates and strings) in the WHERE clause must be enclosed in single quotes. Examples are shown below.

Code Sample:

```
    /*
Create a report showing the title and rating
of all films where rating is PG
*/

SELECT title, rating
FROM film
WHERE rating = 'PG';
```

Code Explanation

Show the title and rating of all films where rating is PG

Code Sample:

```
    /*
Create a report showing the first and last name of all customers
that do not belong to store 2.
*/

SELECT store_id, first_name, last_name
FROM customer
```

```
WHERE store_id <> 2;
```

Code Explanation

Create a report showing the first and last name of all customers that do not belong to store 2.

Code Sample: BasicSelects/Demos/Where-GreaterThanOrEqual.sql

```
SELECT title, rating, rental_rate
FROM film
WHERE length >= 180;
```

Code Explanation

Checking for films that run more than 3 hours

Exercise: Using the WHERE clause to check for equality or inequality

Duration: 5 to 10 minutes.

In this exercise, you will practice using the **WHERE clause** to check for equality and inequality.

1. Create a report showing last and first names all the customes of *sakila* associated with Store ID=1.
2. Create a report showing the title, rating and rental rate of all films with rental rate greater than 2.
3. Create a report showing the title, rating and rental rate of all films in "Italian" language. This may require locating the language Italian first.

Checking for NULL

When a field in a row has no value, it is said to be NULL. This is not the same as having an empty string. Rather, it means that the field contains no value at all. When checking to see if a field is NULL, you cannot use the equals sign (=); rather, use the IS NULL expression.

Code Sample:

```
      /*
Create a report showing films where category is not defined.
*/

SELECT title, rating, rental_rate
FROM film_detail
WHERE category_name IS NULL;
```

Code Explanation

Create a report showing films where category is not defined.

Code Sample:

```
      SELECT last_name, first_name
FROM staff
WHERE picture IS NOT NULL;
```

Code Explanation

Select staff where picture is not null

Exercise: Checking for NULL

Duration: 5 to 10 minutes.

In this exercise, you will practice selecting records with fields with NULL values. We will work with the *film_detail* table for this exercise.

1. Create a report that shows the title, rating, length of all films that don't have a description.
2. Create a report that shows the title, rating, length of all films that have a language specified.

WHERE and ORDER BY

When using WHERE and ORDER BY together, the **WHERE clause** must come before the **OR DER BY clause** .

Code Sample:

```
    /*
Create a report showing the first and last name of all customers whose
last names start with a letter in the last half of the alphabet.
Sort by Last Name in descending order.
*/

SELECT first_name, last_name
FROM customer
WHERE last_name >= 'N'
ORDER BY last_name DESC;
```

Code Explanation

Select customers with a name condition ordered by last name

Exercise: Using WHERE and ORDER BY Together

Duration: 5 to 10 minutes.

In this exercise, you will practice writing SELECT statements that use both WHERE and ORDE
R BY .
We will work with the
film_detail
table for this exercise.

1. Create a report that shows title, rating, category, price, length of films of 'Comedy' category ordered by length.
2. Create a report that shows title, rating, category, price, length of films that are longer than 3 hours sorted by decreasing rental rate.

Using Aliases

The column names and the table name can be followed by an optional AS subclause that allows you to assign an
alias
to these names.

Code Sample:

```
      SELECT fl.title AS 'Film Title', fl.rating AS 'Film Rating'
FROM film AS fl
WHERE fl.film_id = 998;
```

Code Explanation

Select film data with table and column **aliases** for a given film

Selecting Distinct Records

The **DISTINCT keyword** is used to select distinct combinations of column values from a table. For example, the following example selects distinct ratings assigned to existing films.

Code Sample:

```
      /*
Find all the distinct ratings assigned to existing films.
*/

SELECT DISTINCT rating
FROM film;
```

The following example counts unique or DISTINCT customers who have rented movies.

Code Sample:

```
      /*
How many distinct customers have rented movies.
*/

SELECT COUNT(DISTINCT customer_id) AS 'Customer Count'
FROM rental;

SELECT COUNT(customer_id) FROM customer;
```

Sorting Records

The **ORDER BY clause** of the **SELECT statement** is used to sort records. The **ORDER BY clause**

determines the order in which rows are returned in a results set. The following syntax describes the elements in an
ORDER BY
clause
:

```
    ORDER BY <order by definition>
<order by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
```

As the syntax indicates, the **ORDER BY clause** must include the **ORDER BY keywords** and at least one column name or a column alias. If you include more than one column, a comma must separate them.

By default, for each column in an **ORDER BY clause**, records are sorted in ascending order. This can be explicitly specified with the A
SC
keyword. To sort records in descending order, use the
DESC
keyword
after the column name. In addition, when more than one column is specified, the rows are sorted first by the column that is listed first, then by the next specified column, and so on.

Several variations of ORDER BY are shown here:

Code Sample:

```
      SELECT customer_id, store_id, first_name, last_name
FROM customer
ORDER BY last_name;
```

Code Explanation

Sorting by a single column *last_name*

Code Sample:

```
      SELECT customer_id, store_id, first_name, last_name
FROM customer
ORDER BY store_id, last_name;
```
Code Explanation

Nested sorting By Multiple Columns.

Code Sample:

```
      SELECT film_id, title, rating, rental_rate
FROM film
ORDER BY 4 ASC, 3 DESC;
```
Code Explanation

Sorting By Column Position (numeric) instead of column name.

You can sort your result sets by as many columns as it is practical; however, this is useful only if the columns listed first have enough repeated values to make sorting additional columns return meaningful results.

Code Sample:

```
      SELECT title, rating, rental_rate
FROM film
ORDER BY rental_rate ASC, rating DESC, title ASC;
```
Code Explanation

Specify and mix Ascending and Descending Sorts.

In this case, the returned rows are sorted first by the values in the rental_rate column in ascending order) and then by the values in the rating column in descending order and so on.

Exercise: Sorting Results

Duration: 5 to 10 minutes.

In this exercise, you will practice sorting results in SELECT statements.

1. Select name from category sorted by name.
2. Select last_name, first_name, email from the customer table sorted by email.
3. Select title, rating, rental_rate, length from film_detail where rental_rate > 3 sorted by rating and length DESC

The LIMIT Clause

We now review the **LIMIT clause**. The **LIMIT clause** is used most effectively in a **SELECT statement**
when it is used with an
ORDER BY
clause
.

The **LIMIT clause** takes two arguments, as the following syntax shows:

```
LIMIT [<offset>,] <row count>
```

The first option, <offset>, is optional and indicates where to begin the LIMIT row count. If no value is specified, 0 is assumed. (The first row in a result set is considered to be 0, rather than 1.) The second argument,
<row count>
in the
LIMIT
clause, indicates the number of rows to be returned. For example, the following
SELECT
statement includes a
LIMIT
clause
that specifies a row count of 5.

Code Sample:

```
    SELECT last_name, first_name, email
FROM customer
LIMIT 5;
```

Code Explanation

Select any first five customers.

Code Sample:

```
      SELECT title, rating, rental_rate
FROM film
ORDER BY rating DESC, title ASC LIMIT 5;
```

Code Explanation

As no offset value is specified, so 0 is assumed. The row count value is specified as 5, so the first five rows of the result set are returned.

Select limited number of ordered rows

If you were to specify an offset value, the rows returned would begin with the first row indicated by the offset value and end after the number of rows indicated by the row count is returned. For example, the following SELECT statement includes a LIMIT clause that specifies an offset value of 2 and a row count value of 3:

Code Sample:

```
      SELECT title, rating, rental_rate
FROM film
ORDER BY rental_rate ASC, rating DESC, title ASC LIMIT 2,4;
```

Code Explanation

The example includes a LIMIT clause that specifies an offset value of 2 and a row count value of 3. So three rows starting from the third row are returned.

Select limited number of ordered rows from an offset

Processing the First N or Last N Records

Using LIMIT in combination with ORDER BY allows us to construct a list of the *Top N*, which means the the first N, the best, the worst, the oldest, or the most useless or meaningful rows in a target data set.

ORDER BY gives us a sorting criterion on expressions. If you want not the first, but the last results, then reverse the sort order with DESC.

In subsequent lessons, we will combine these clauses with Aggregation and multi-table joins to get to top or the bottom of entities more accurately.

Basic Select Statements Conclusion

In this lesson of the MySQL tutorial, you have learned a lot about basic data retrieval with SELE CT .
However, this is just the tip of the iceberg.
SELECT
statements can get a lot more powerful and, of course, a lot more complicated.

As the lesson has demonstrated, the SELECT statement can contain many components, allowing you to create statements as simple or as complex as necessary to retrieve specific data from the tables in your MySQL database. You can specify which columns to retrieve, which rows to retrieve, how the rows should be sorted, whether the rows should be grouped together and summarized, and the number of rows to return in your result set. To provide you with the information you need to create robust SELECT st atements, the lesson gave you the information necessary to perform the following tasks:

- Create **SELECT statements** that retrieve all columns and all rows from a table.
- Create **SELECT statements** that retrieve specific columns from a table.
- Assign **aliases** to table and column names
- Add options to your **SELECT statements**.
- Add **WHERE clauses** to your **SELECT statements** that determine which rows the statements would return.
- Add **ORDER BY clauses** to your **SELECT statements** to sort the rows returned by your statements
- Add **LIMIT clauses** to your **SELECT statement** to limit the number of rows returned by the statement

In later lessons, you will learn how to refine your **SELECT statements** even further. For example, you learn how to join tables in a **SELECT statement** or to embed other **SELECT statements** in a parent **SELECT statement**. In addition, you also learn more about using operators and functions to create powerful expressions in your statements.

To continue to **learn MySQL** go to the top of this page and click on the next **lesson in this MySQL** Tutorial's Table of Contents.