PHP Tutorial

PHP enables you to create dynamic web pages, develop websites, and generate dynamic content. You'll also be able to create contact forms, forums, blogs, picture galleries, surveys, social networks, and a whole lot more.
PHP: Hypertext Preprocessor (**PHP**) is a free, highly popular, open source scripting language. PHP scripts are executed on the **server**.

Just a short list of what PHP is capable of:

- Generating dynamic page content
- Collecting form data
- Adding, deleting, and modifying information stored in your database
- Creating, opening, reading, writing, deleting, and closing files on the server
- and much more!

PHP has enough power to work at the core of **WordPress**, the busiest blogging system on the web. It also has the degree of depth required to run **Facebook**, the web's largest social network!

PHP is a Server side programming language

Why PHP

PHP **runs** on numerous, varying platforms, including Windows, Linux, Unix, Mac OS X, and so on.
PHP is **compatible** with almost any modern server, such as Apache, IIS, and more.
PHP **supports** a wide range of databases.
PHP is **free**!

PHP Syntax

A PHP script starts with **<?php** and ends with **?>**:

```
<?php
   // PHP code goes here
?>
```

Echo

PHP has a built-in "echo" function, which is used to output text.
In actuality, it's not a function; it's a **language construct**. As such, it does not require parentheses.

Let's output a text.

```
<?php
   echo "I love PHP!";
?>
```

The text should be in single or double **quotation marks**.

PHP Syntax

You can also use the shorthand PHP tags, <? ?>, as long as they're supported by the server.

```
<?
   echo "Hello World!";
?>
```

However, **<?php ?>**, as the official standard, is the recommended way of defining PHP scripts.

<?php … ?> is the most widely recommended way to use PHP tags?

PHP Statements

Each PHP statement must end with a **semicolon**.

```
<?
   echo "A";
   echo "B";
   echo "C";
?>
```

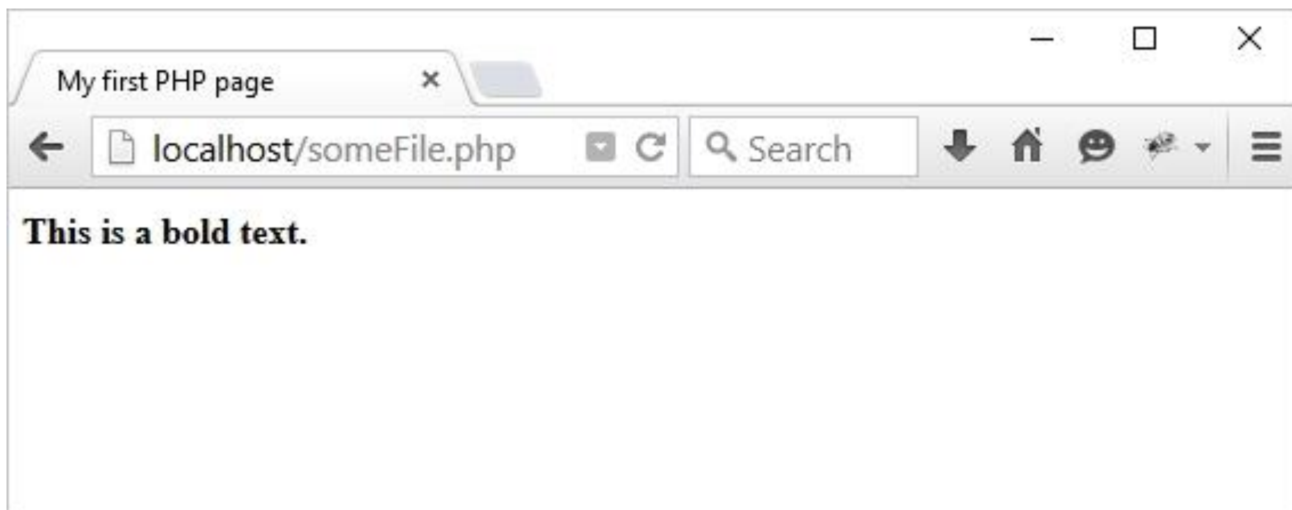
PHP statements end with **semicolons (;)**.
Forgetting to add a semicolon at the end of a statement results in an **error**.

Echo

HTML markup can be added to the text in the **echo** statement.

```
<?php
    echo "<strong>This is a bold text.</strong>";
?>
```

Result:



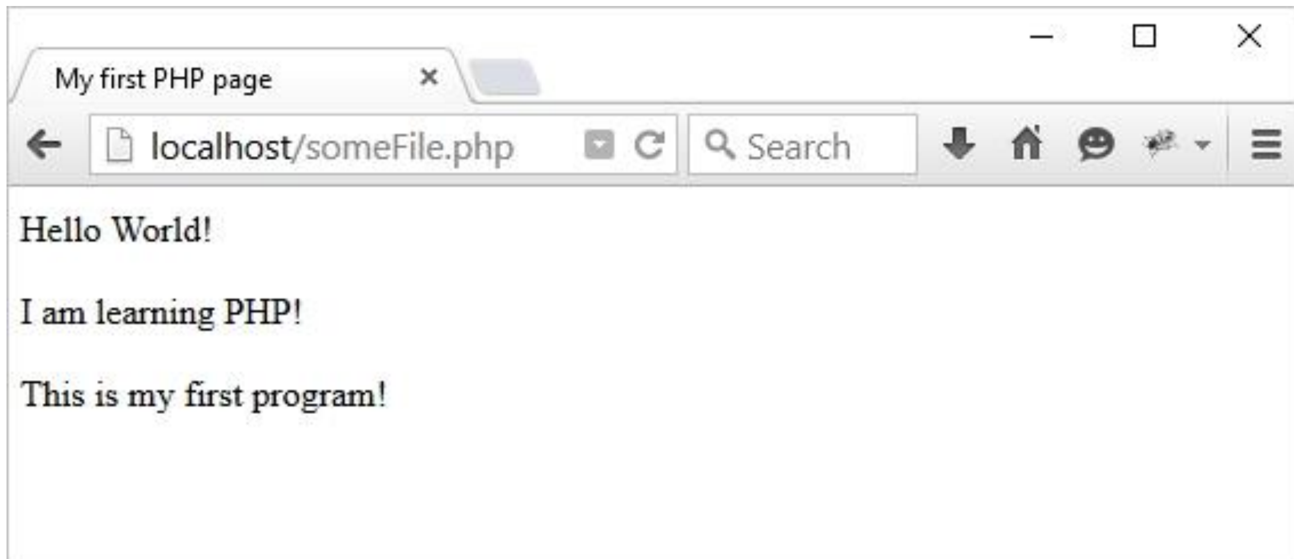
Comments

In PHP code, a **comment** is a line that is not executed as part of the program. You can use comments to communicate to others so they understand what you're doing, or as a reminder to yourself of what you did.

A **single-line** comment starts with //:

```
<?php
    echo "<p>Hello World!</p>";
    // This is a single-line comment
    echo "<p>I am learning PHP!</p>";
    echo "<p>This is my first program!</p>";
?>
```

Result:



Multi-Line Comments

Multi-line comments are used for composing comments that take more than a single line.
A multi-line comment begins with /* and ends with */.

```
<?php
    echo "<p>Hello World!</p>";
    /*
    This is a multi-line comment block
    that spans over
    multiple lines
    */
    echo "<p>I am learning PHP!</p>";
    echo "<p>This is my first program!</p>";
?>
```

Adding comments as you write your code is a good practice. It helps others understand your thinking and makes it easier for you to recall your thought processes when you refer to your code later on.

Variables

Variables are used as "containers" in which we store information.
A PHP variable starts with a dollar sign ($), which is followed by the name of the variable.
`$variable_name = value;`

Rules for PHP variables:

- A variable name must start with a letter or an underscore
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive ($name and $NAME would be two different variables)

For example:

```
<?php
    $name = 'John';
    $age = 25;
    echo $name;  //Outputs 'John'
?>
```

In the example above, notice that we did not have to tell PHP which data type the variable is.
PHP automatically converts the variable to the correct data type, depending on its value.

Unlike other programming languages, PHP has no command for declaring a variable. It is created the moment you first assign a value to it.

Constants

Constants are similar to variables except that they cannot be changed or undefined after they've been defined.
Begin the name of your constant with a letter or an underscore.
To create a constant, use the **define()** function:
`define(name, value)`

Parameters:
name: Specifies the name of the constant;
value: Specifies the value of the constant;

The example below creates a constant:

```
<?php
    define("MSG", "Hello World!");
    echo MSG;  // Outputs "Hello World!"
?>
```

No dollar sign ($) is necessary before the constant name.

Data Types

Variables can store a variety of data types.
Data types supported by PHP: **String**, **Integer**, **Float**, **Boolean**, **Array**, **Object**, NULL, Resource.

PHP String

A string is a sequence of characters, like "Hello world!"
A string can be any text within a set of single or double **quotes**.

```
<?php
    $string1 = "Hello world!"; //double quotes
    $string2 = 'Hello world!'; //single quotes
?>
```

You can join two strings together using the dot (.) concatenation operator.
For example: echo **$s1 . $s2**

PHP Integer

An integer is a whole number (without decimals) that must fit the following criteria:
- It cannot contain commas or blanks
- It must not have a decimal point
- It can be either positive or negative

```
<?php
    $int1 = 42; // positive number
    $int2 = -42; // negative number
?>
```

PHP Boolean

A **Boolean** represents two possible states: TRUE or FALSE.

```
<?php
    $x = true; $y = false;
?>
```

Booleans are often used in conditional testing.

Most of the data types can be used in combination with one another.
In this example, string and integer are put together to determine the sum of two numbers.

```
<?php
    $str = "10";
    $int = 20;
    $sum = $str + $int;
    echo ($sum);  // Outputs 30
?>
```

PHP automatically converts each variable to the correct data type, according to its value. This is why the variable **$str** is treated as a number in the addition.

Variables Scope

PHP variables can be declared anywhere in the script.
The **scope** of a variable is the part of the script in which the variable can be referenced or used.

PHP's most used variable scopes are **local**, **global**.
A variable declared outside a function has a **global scope**.
A variable declared within a function has a **local scope**, and can only be accessed within that function.

Consider the following example.

```
<?php
    $name = 'David';
    function getName()
    {
        echo $name;
    }

    getName();
    // Error: Undefined variable: name
?>
```

This script will produce an error, as the **$name** variable has a **global** scope, and is not accessible within the **getName()** function.

Arrays

An **array** is a special variable, which can hold more than one value at a time.
If you have a list of items (a list of names, for example), storing them in single variables would look like this:

```
$name1 = "David";
$name2 = "Amy";
$name3 = "John";
```

But what if you have 100 names on your list? The solution: Create an **array**!

Numeric Arrays

Numeric or indexed arrays associate a numeric index with their values.
The index can be assigned automatically (index always starts at **0**), like this:

```
$names = array("David", "Amy", "John");
```

As an alternative, you can assign your index manually.

```
$names[0] = "David";
$names[1] = "Amy";
$names[2] = "John";
```

We defined an array called **$names** that stores three values.
You can access the array elements through their indices.

```
echo $names[1]; // Outputs "Amy"
```

Remember that the first element in an array has the index of **0**, not 1.

Numeric Arrays

You can have integers, strings, and other data types together in one array.
Example:

```
<?php
    $myArray[0] = "John";
    $myArray[1] = "<strong>PHP</strong>";
    $myArray[2] = 21;

    echo "$myArray[0] is $myArray[2] and knows $myArray[1]";

    // Outputs "John is 21 and knows PHP"
?>
```

Associative Arrays

Associative arrays are arrays that use named keys that you assign to them.
There are two ways to create an associative array:

```
$people = array("David"=>"27", "Amy"=>"21", "John"=>"42");
// or
$people['David'] = "27";
$people['Amy'] = "21";
$people['John'] = "42";
```

In the first example, note the use of the **=>** signs in assigning values to the named keys.

Use the named keys to access the array's members.

```
$people = array("David"=>"27", "Amy"=>"21", "John"=>"42");
echo $people['Amy']; // Outputs 21"
```

Conditional Statements

Conditional statements perform different actions for different decisions.
The **if else** statement is used to execute a certain code if a condition is **true**, and another code if the condition is **false**.
Syntax:

```
if (condition) {
    code to be executed if condition is true;
} else {
    code to be executed if condition is false;
}
```

You can also use the **if** statement without the **else** statement, if you do not need to do anything, in case the condition is **false**.

If Else

The example below will output the greatest number of the two.

```
<?php
    $x = 10;
    $y = 20;
    if ($x >= $y)
    {
        echo $x;
    }
    else
    {
        echo $y;
    }

    // Outputs "20"
?>
```

The Elseif Statement

Use the **if...elseif...else** statement to specify a **new** condition to test, if the first condition is **false**.

Syntax:

```
if (condition)
{
   code to be executed if condition is true;
}
elseif (condition)
{
    code to be executed if condition is true;
}
else
{
   code to be executed if condition is false;
}
```

You can add as many **elseif** statements as you want. Just note, that the **elseif** statement must begin with an **if** statement.

The Elseif Statement

For example:

```
<?php
    $age = 21;

    if ($age<=13)
    {
        echo "Child.";
    }
    elseif ($age>13 && $age<19)
    {
        echo "Teenager";
    }
    else
    {
        echo "Adult";
    }

    //Outputs "Adult"
?>
```

We used the **logical AND** (&&) operator to combine the two conditions and check to determine whether $age is between 13 and 19.

Loops

When writing code, you may want the same block of code to run over and over again. Instead of adding several almost equal code-lines in a script, we can use **loops** to perform a task like this.

The while Loop

The **while** loop executes a block of code as long as the specified condition is **true**.

Syntax:

```
while (condition is true)
{
    code to be executed;
}
```

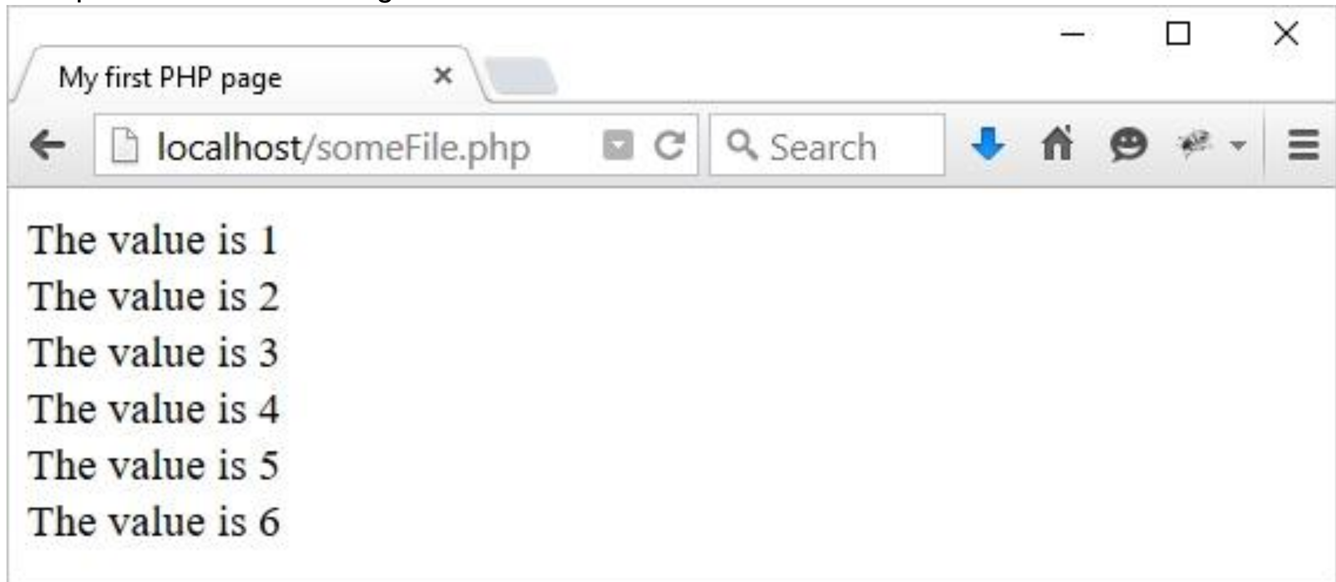
If the condition never becomes false, the statement will continue to execute indefinitely.

The while Loop

The example below first sets a variable $i to one ($i = 1). Then, the **while** loop runs as long as $i is less than seven ($i < 7). $i will increase by one each time the loop runs ($i++):

```
$i = 1;
while ($i < 7)
{
    echo "The value is $i <br/>";
    $i++;
}
```

This produces the following result:



The do...while Loop

The **do...while** loop will always execute the block of code once, check the condition, and repeat the loop as long as the specified condition is true.

Syntax:

```
do
{
    code to be executed;
} while (condition is true);
```

Regardless of whether the condition is **true** or **false**, the code will be executed at least **once**, which could be needed in some situations.

The do...while Loop

The example below will write some output, and then increment the variable $i by one. Then the condition is checked, and the loop continues to run, as long as $i is less than or equal to 7.

```
$i = 5;
do
{
    echo "The number is " . $i . "<br/>";
    $i++;
} while($i <= 7);

//Output
//The number is 5
//The number is 6
//The number is 7
```

Note that in a **do while** loop, the condition is tested AFTER executing the statements within the loop. This means that the **do while** loop would execute its statements at least once, even if the condition is false the first time.

The for Loop

The **for** loop is used when you know in advance how many times the script should run.

```
for (init; test; increment)
{
    code to be executed;
}
```

Parameters:
init: Initialize the loop counter value
test: Evaluates each time the loop is iterated, continuing if evaluates to **true**, and ending if it evaluates to **false**
increment: Increases the loop counter value

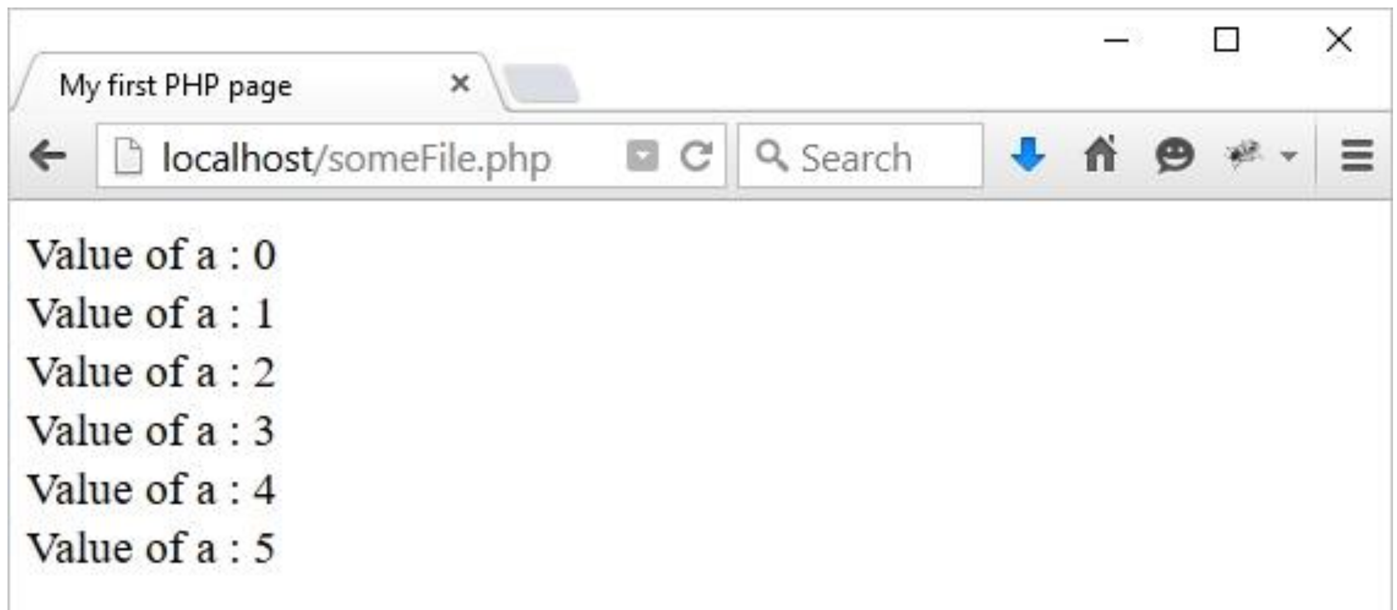
Each of the parameter expressions can be empty or contain multiple expressions that are separated with **commas**.
In the **for** statement, the parameters are separated with **semicolons**.

The for Loop

The example below displays the numbers from 0 to 5:

```
for ($a = 0; $a < 6; $a++) {
    echo "Value of a : ". $a . "<br/>";
}
```

Result:



The **for** loop in the example above first sets the variable **$a** to 0, then checks for the condition ($a < 6). If the condition is true, it runs the code. After that, it increments $a ($a++).

The foreach Loop

The **foreach** loop works only on arrays, and is used to loop through each key/value pair in an array.
There are two syntaxes:

```
foreach (array as $value)
{
    code to be executed;
}
//or
foreach (array as $key => $value)
{
    code to be executed;
}
```

The first form loops over the array. On each iteration, the value of the current element is assigned to **$value**, and the array pointer is moved by one, until it reaches the last array element.
The second form will additionally assign the current element's key to the **$key** variable on each iteration.

The following example demonstrates a loop that outputs the values of the **$names** array.

```
$names = array("John", "David", "Amy");
foreach ($names as $name)
{
    echo $name.'<br/>';
}

// John
// David
// Amy
```

The switch Statement

The **switch** statement is an alternative to the **if-elseif-else** statement.
Use the **switch** statement to select one of a number of blocks of code to be executed.

Syntax:

```
switch (n)
{
    case value1:
        //code to be executed if n=value1
    break;
    case value2:
        //code to be executed if n=value2
    break;
    ...
    default:
        // code to be executed if n is different from all labels
}
```

First, our single expression, **n** (most often a variable), is evaluated once. Next, the value of the expression is compared with the value of each case in the structure. If there is a match, the block of code associated with that case is executed.

Using **nested if else statements** results in similar behavior, but switch offers a more elegant and optimal solution.

Switch

Consider the following example, which displays the appropriate message for each day.

```
$today = 'Tue';

switch ($today)
{
    case "Mon":
        echo "Today is Monday.";
    break;
    case "Tue":
        echo "Today is Tuesday.";
    break;
    case "Wed":
        echo "Today is Wednesday.";
    break;
    case "Thu":
        echo "Today is Thursday.";
    break;
    case "Fri":
        echo "Today is Friday.";
    break;
    case "Sat":
        echo "Today is Saturday.";
    break;
    case "Sun":
        echo "Today is Sunday.";
    break;
    default:
        echo "Invalid day.";
}
//Outputs "Today is Tuesday."
```

The **break** keyword that follows each case is used to keep the code from automatically running into the next case. If you forget the **break;** statement, PHP will automatically continue through the next case statements, even when the case doesn't match.

default

The **default** statement is used if no match is found.

```
$x=5;
switch ($x)
{
    case 1:
        echo "One";
    break;
    case 2:
        echo "Two";
    break;
    default:
        echo "No match";
}
//Outputs "No match"
```

The **default** statement is optional, so it can be omitted.

Switch

Failing to specify the **break** statement causes PHP to continue to executing the statements that follow the **case**, until it finds a **break**. You can use this behavior if you need to arrive at the same output for more than one case.

```
$day = 'Wed';

switch ($day)
{
    case 'Mon':
        echo 'First day of the week';
    break;
    case 'Tue':
    case 'Wed':
    case 'Thu':
        echo 'Working day';
    break;
    case 'Fri':
        echo 'Friday!';
    break;
    default:
        echo 'Weekend!';
}
//Outputs "Working day"
```

The example above will have the same output if **$day** equals 'Tue', 'Wed', or 'Thu'.

The break Statement

As discussed in the previous lesson, the **break** statement is used to break out of the **switch** when a case is matched.
If the break is absent, the code keeps running. For example:

```
$x=1;
switch ($x)
{
    case 1:
        echo "One";
    case 2:
        echo "Two";
    case 3:
        echo "Three";
    default:
        echo "No match";
}

//Outputs "OneTwoThreeNo match"
```

Break can also be used to halt the execution of **for**, **foreach**, **while**, **do-while** structures.

The **break** statement ends the current **for**, **foreach**, **while**, **do-while** or **switch** and continues to run the program on the line coming up after the loop.
A **break** statement in the outer part of a program (e.g., not in a control loop) will stop the script.

The continue Statement

When used within a looping structure, the **continue** statement allows for skipping over what remains of the current loop iteration. It then continues the execution at the condition evaluation and moves on to the beginning of the next iteration.

The following example skips the even numbers in the **for** loop:

```
for ($i=0; $i<10; $i++)
{
    if ($i%2==0)
    {
        continue;
    }
    echo $i . ' ';
}

//Output: 1 3 5 7 9
```

You can use the **continue** statement with all looping structures.

include

The **include** and **require** statements allow for the insertion of the content of one PHP file into another PHP file, before the server executes it.
Including files saves quite a bit of work. You can create a standard header, footer, or menu file for all of your web pages. Then, when the header is requiring updating, you can update the header include file only.

Assume that we have a standard header file called **header.php**.

```
<?php
    echo '<h1>Welcome</h1>';
?>
```

Use the **include** statement to include the header file in a page.

```
<html>
<body>

    <?php include 'header.php'; ?>

    <p>Some text.</p>
    <p>Some text.</p>
</body>
</html>
```

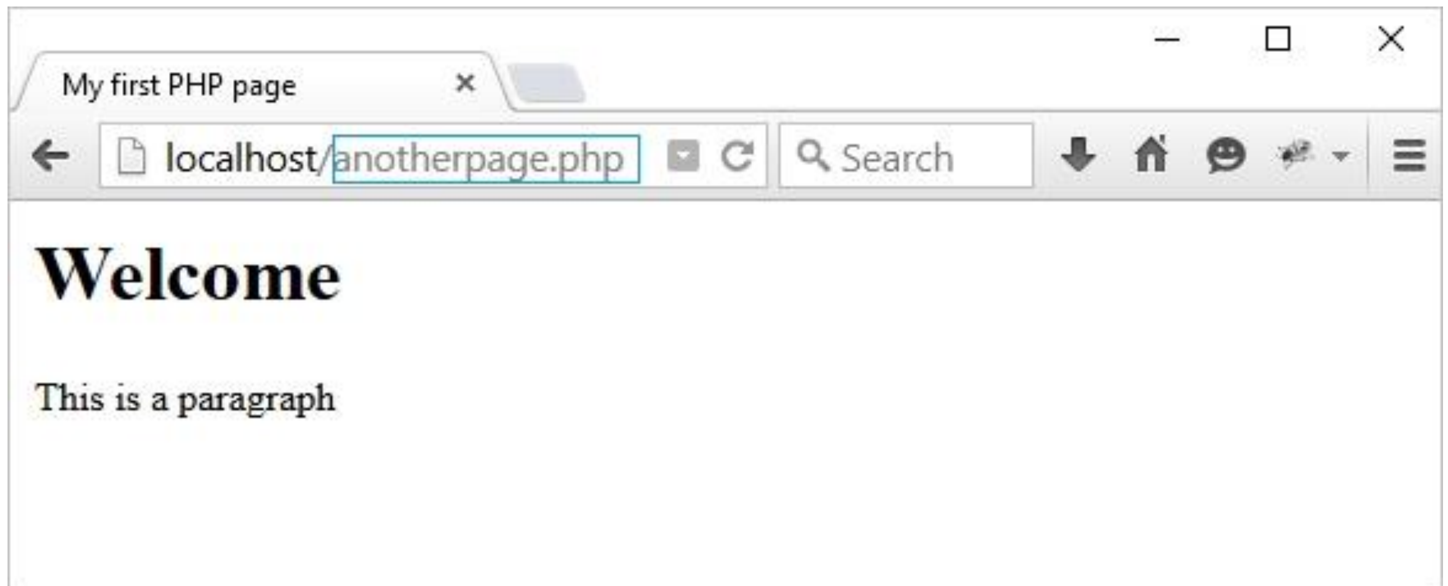
include

Using this approach, we have the ability to include the same **header.php** file into multiple pages.

```
<html>
<body>
    <?php include 'header.php'; ?>

    <p>This is a paragraph</p>
</body>
</html>
```

Result:



Files are included based on the file **path**.
You can use an **absolute** or a **relative** path to specify which file should be included.

include vs require

The **require** statement is identical to include, the exception being that, upon failure, it produces a fatal error.
When a file is included using the **include** statement, but PHP is unable to find it, the script continues to execute.
In the case of **require**, the script will cease execution and produce an error.

Use **require** when the file is required for the application to run.
Use **include** when the file is not required. The application should continue, even when the file is not found.

Functions

A **function** is a block of statements that can be used repeatedly in a program.
A function will not execute immediately when a page loads. It will be executed by a call to the function.
A user defined function declaration starts with the word **function**:

```
function functionName()
{
    //code to be executed
}
```

A function name can start with a letter or an underscore, but not with a number or a special symbol.

Function names are NOT case-sensitive.

Functions

In the example below, we create the function **sayHello()**. The opening curly brace ({) indicates that this is the beginning of the function code, while the closing curly brace (}) indicates that this is the end.
To call the function, just write its name:

```
function sayHello()
{
    echo "Hello!";
}

sayHello(); //call the function

//Outputs "Hello!"
```

Function Parameters

Information can be passed to functions through **arguments**, which are like variables.
Arguments are specified after the function name, and within the parentheses.
Here, our function takes a number, multiplies it by two, and prints the result:

```
function multiplyByTwo($number)
{
    $answer = $number * 2;
    echo $answer;
}

multiplyByTwo(3);  //Outputs 6
```

You can add as many arguments as you want, as long as they are separated with **commas**.

```
function multiply($num1, $num2)
{
    echo $num1 * $num2;
}


multiply(3, 6);  //Outputs 18
```

When you define a function, the variables that represent the values that will be passed to it for processing are called **parameters**. However, when you use a function, the value you pass to it is called an **argument**.

Function Parameters

Information can be passed to functions through **arguments**, which are like variables.
Arguments are specified after the function name, and within the parentheses.
Here, our function takes a number, multiplies it by two, and prints the result:

```
function multiplyByTwo($number)
{
    $answer = $number * 2;
    echo $answer;
}

multiplyByTwo(3); //Outputs 6
```

You can add as many arguments as you want, as long as they are separated with **commas**.

```
function multiply($num1, $num2)
{
    echo $num1 * $num2;
}

multiply(3, 6);  //Outputs 18
```

When you define a function, the variables that represent the values that will be passed to it for processing are called **parameters**. However, when you use a function, the value you pass to it is called an **argument**.

Return

A function can return a value using the **return** statement.
Return stops the function's execution, and sends the value back to the calling code.
For example:

```
function mult($num1, $num2)
{
    $res = $num1 * $num2;
    return $res;
}

echo mult(8, 3);  // Outputs 24
```

Leaving out the return results in a **NULL** value being returned.
A function cannot return multiple values, but returning an **array** will produce similar results.

Predefined Variables

A **superglobal** is a predefined variable that is always accessible, regardless of scope. You can access the PHP superglobals through any function, class, or file.

PHP's superglobal variables are $_SERVER, $GLOBALS, $_REQUEST, $_POST, $_GET, $_FILES, $_ENV, $_COOKIE, $_SESSION.

$_SERVER

$_SERVER is an array that includes information such as headers, paths, and script locations. The entries in this array are created by the web server.
$_SERVER['SCRIPT_NAME'] returns the path of the current script:

```
<?php
    echo $_SERVER['SCRIPT_NAME'];
    //Outputs "/somefile.php"
?>
```

This graphic shows the main elements of $_SERVER.

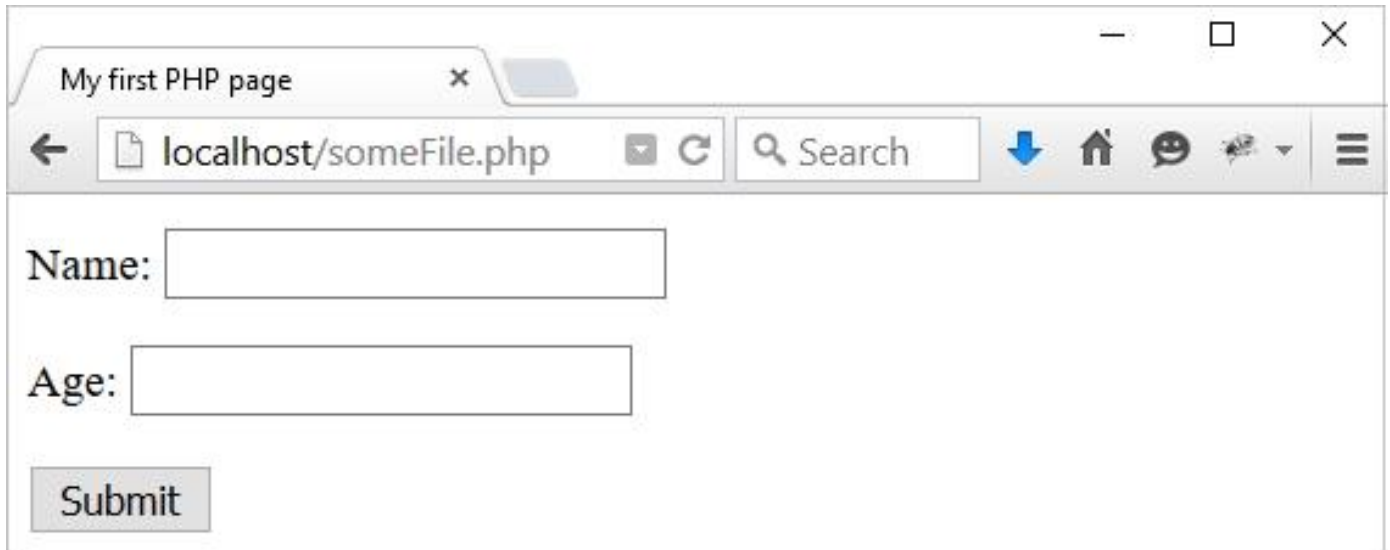
Element/Code	Description
$_SERVER['PHP_SELF']	Returns the filename of the currently executing script
$_SERVER['SERVER_ADDR']	Returns the IP address of the host server
$_SERVER['SERVER_NAME']	Returns the name of the host server
$_SERVER['HTTP_HOST']	Returns the Host header from the current request
$_SERVER['REMOTE_ADDR']	Returns the IP address from where the user is viewing the current page
$_SERVER['REMOTE_HOST']	Returns the Host name from where the user is viewing the current page
$_SERVER['REMOTE_PORT']	Returns the port being used on the user's machine to communicate with the web server
$_SERVER['SCRIPT_FILENAME']	Returns the absolute pathname of the currently executing script
$_SERVER['SERVER_PORT']	Returns the port on the server machine being used by the web server for communication (such as 80)
$_SERVER['SCRIPT_NAME']	Returns the path of the current script
$_SERVER['SCRIPT_URI']	Returns the URI of the current page

Forms

The purpose of the PHP superglobals **$_GET** and **$_POST** is to collect data that has been entered into a form.
The example below shows a simple HTML form that includes two input fields and a submit button:

```
<form action="first.php" method="post">
    <p>Name: <input type="text" name="name" /></p>
    <p>Age: <input type="text" name="age" /></p>
    <p><input type="submit" name="submit" value="Submit" /></p>
</form>
```

Result:



Forms

The **action** attribute specifies that when the form is submitted, the data is sent to a PHP file named **first.php**.
HTML form elements have **names**, which will be used when accessing the data with PHP.
The **method** attribute is set to the value to "**post**".

Forms

Now, when we have an HTML form with the **action** attribute set to our PHP file, we can access the posted form data using the **$_POST** associative array.
In the **first.php** file:

```
<html>
<body>
    Welcome <?php echo $_POST["name"]; ?><br/>
    Your age: <?php echo $_POST["age"]; ?>
</body>
</html>
```

The **$_POST** superglobal array holds key/value pairs. In the pairs, keys are the **names** of the form controls and values are the **input data** entered by the user.

We used the **$_POST** array, as the **method="post"** was specified in the form.

POST

The two methods for submitting forms are **GET** and **POST**.
Information sent from a form via the **POST** method is invisible to others, since all names and/or values are embedded within the body of the HTTP request. Also, there are no limits on the amount of information to be sent.
However, it is not possible to bookmark the page, as the submitted values are not visible.

POST is the preferred method for sending form data.

GET

Information sent via a form using the **GET** method is visible to everyone (all variable names and values are displayed in the **URL**). **GET** also sets limits on the amount of information that can be sent - about 2000 characters (depending on the browser).
However, because the variables are displayed in the URL, it is possible to bookmark the page, which can be useful in some situations.

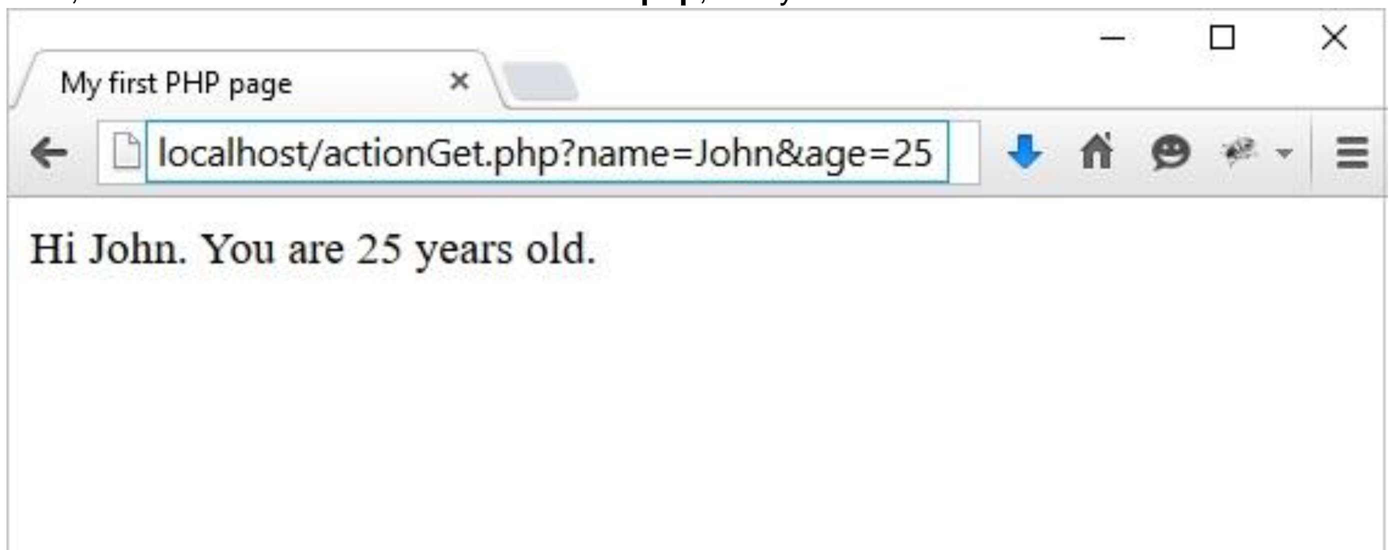
For example:

```
<form action="actionGet.php" method="get">
    Name: <input type="text" name="name" /><br/><br/>
    Age: <input type="text" name="age" /><br/><br/>
    <input type="submit" name="submit" value="Submit" />
</form>
```

actionGet.php

```
<?php
    echo "Hi ".$_GET['name'].". ";
    echo "You are ".$_GET['age']." years old.";
?>
```

Now, the form is submitted to the **actionGet.php**, and you can see the submitted data in the **URL**:



GET should **NEVER** be used for sending passwords or other sensitive information!
When using POST or GET, proper validation of form data through filtering and processing is vitally important to protect your form from hackers and exploits!

Cookies

Cookies are often used to identify the user. A cookie is a small file that the server embeds on the user's computer. Each time the same computer requests a page through a browser, it will send the cookie, too. With PHP, you can both create and retrieve cookie values.

Create cookies using the **setcookie()** function:
`setcookie(name, value, expire);`

name: Specifies the cookie's name
value: Specifies the cookie's value
expire: Specifies (in seconds) when the cookie is to expire. The value: time()+86400*30, will set the cookie to expire in 30 days. If this parameter is omitted or set to 0, the cookie will expire at the end of the session (when the browser closes). Default is 0.

The following example creates a cookie named "user" with the value "John". The cookie will expire after 30 days, which is written as 86,400 * 30, in which 86,400 seconds = one day.

We then retrieve the value of the cookie "user" (using the global variable **$_COOKIE**). We also use the **isset()** function to find out if the cookie is set:

```
<?php
    $value = "John";
    setcookie("user", $value, time() + (86400 * 30));

    if(isset($_COOKIE['user']))
    {
        echo "Value is: ". $_COOKIE['user'];
    }
    //Outputs "Value is: John"
?>
```

The **setcookie()** function must appear BEFORE the <html> tag.
The value of the cookie is automatically encoded when the cookie is sent, and is automatically decoded when it's received. Nevertheless, **NEVER** store sensitive information in cookies.

Sessions

Using a **session**, you can store information in variables, to be used across multiple pages.
Information is not stored on the user's computer, as it is with **cookies**.
By default, session variables last until the user closes the browser.

Start a PHP Session

A session is started using the **session_start()** function.
Use the PHP global **$_SESSION** to set session variables.

```
<?php
    // Start the session
    session_start();

    $_SESSION['color'] = "red";
    $_SESSION['name'] = "John";
?>
```

Now, the **color** and **name** session variables are accessible on multiple pages, throughout the entire session.

The **session_start()** function must be the very first thing in your document. Before any HTML tags.

Session Variables

Another page can be created that can access the session variables we set in the previous page:

```
<?php
    // Start the session
    session_start();
?>
<!DOCTYPE html>
<html>
<body>
    <?php
        echo "Your name is " . $_SESSION['name'];
        // Outputs "Your name is John"
    ?>
</body>
</html>
```

Your session variables remain available in the **$_SESSION** superglobal until you close your session.
All global session variables can be removed manually by using **session_unset()**. You can also destroy the session with **session_destroy()**.

A Database

A **database** is a collection of data that is organized in a manner that facilitates ease of access, as well as efficient management and updating.

A database is made up of **tables** that store relevant information.
For example, you would use a database, if you were to create a website like YouTube, which contains a lot of information like videos, usernames, passwords, comments.

Database Tables

A table stores and displays data in a structured format consisting of **columns** and **rows**.
Databases often contain multiple tables, each designed for a specific purpose.
For example, imagine creating a database table of **names** and **telephone numbers**.

First, we would set up columns with the titles *FirstName*, *LastName* and *TelephoneNumber*.

Each table includes its own set of fields, based on the data it will store.

FirstName	LastName	TelephoneNumber
John	Smith	715-555-1230
David	Williams	569-999-1719
Chloe	Anderson	715-777-2010
Emily	Adams	566-333-1223
James	Roberts	763-777-2956

Primary Keys
A primary key is a field in the table that uniquely identifies the table records.

The primary key's main features:
- It must contain a **unique value** for each row.
- It cannot contain **NULL** values.

For example, our table contains a record for each name in a phone book. The unique **ID** number would be a good choice for a primary key in the table, as there is always the chance for more than one person to have the same name.

ID	FirstName	LastName	TelephoneNumber
1	John	Smith	715-555-1230
2	David	Williams	569-999-1719
3	Chloe	Anderson	715-777-2010
4	Emily	Adams	566-333-1223
5	James	Roberts	763-777-2956

- Tables are limited to **ONE** primary key each.
- The primary key's value must be different for each row.

What is SQL?
Once you understand what a database is, understanding SQL is easy.
SQL stands for **S**tructured **Q**uery **L**anguage.

SQL is used to access and manipulate a **database**.
MySQL is a **program** that understands **SQL**.

SQL can:
- insert, update, or delete records in a database.
- create new databases, table, stored procedures, views.
- retrieve data from a database, etc.

SQL is an ANSI (American National Standards Institute) standard, but there are different versions of the SQL language.
Most SQL database programs have their own proprietary extensions in addition to the SQL standard, but all of them support the major commands.

SQL Commands - SELECT Statement

The SELECT statement is used to select data from a database.
The result is stored in a result table, which is called the **result-set**.

A **query** may retrieve information from selected columns or from all columns in the table. To create a simple SELECT statement, specify the name(s) of the column(s) you need from the table.

Syntax of the SQL SELECT Statement:

```
SELECT column_list
FROM table_name
```

- **column_list** includes one or more columns from which data is retrieved
- **table-name** is the name of the table from which the information is retrieved

Below is the data from our **customers** table:

ID	FirstName	LastName	City	ZipCode
1	John	Smith	New York	10199
2	David	Williams	Los Angeles	90052
3	Chloe	Anderson	Chicago	60607
4	Emily	Adams	Houston	77201
5	James	Roberts	Philadelphia	19104

The following SQL statement selects the **FirstName** from the **customers** table:

```
SELECT FirstName
FROM customers
```

FirstName
John
David
Chloe
Emily
James

Remember to end each SQL statement with a **semicolon** to indicate that the statement is complete and ready to be interpreted.
In this tutorial, we will use **semicolon** at the end of each SQL statement.

Case Sensitivity

SQL is case **insensitive**.
The following statements are equivalent and will produce the same result:

```
select City from customers;
SELECT City FROM customers;
sElEct City From customers;
```

It is common practice to write all SQL commands in **upper-case**.

Syntax Rules

A single SQL statement can be placed on one or more text lines. In addition, multiple SQL statements can be combined on a single text line.
For example, the following query is absolutely correct.

```
SELECT City
FROM customers;
```

Combined with proper spacing and indenting, breaking up the commands into logical lines will make your SQL statements much easier to read and maintain.

Selecting Multiple Columns

As previously mentioned, the SQL SELECT statement retrieves records from tables in your SQL database. You can select multiple table columns at once.
Just list the column names, separated by **commas**:

```
SELECT FirstName, LastName, City
FROM customers;
```

Result:

FirstName	LastName	City
John	Smith	New York
David	Williams	Los Angeles
Chloe	Anderson	Chicago
Emily	Adams	Houston
James	Roberts	Philadelphia

Do not put a comma after the **last** column name.

Selecting All Columns

To retrieve all of the information contained in your table, place an **asterisk (*)** sign after the SELECT command, rather than typing in each column names separately.

The following SQL statement selects all of the columns in the **customers** table:

```
SELECT * FROM customers;
```

Result:

ID	FirstName	LastName	City	ZipCode
1	John	Smith	New York	10199
2	David	Williams	Los Angeles	90052
3	Chloe	Anderson	Chicago	60607
4	Emily	Adams	Houston	77201
5	James	Roberts	Philadelphia	19104

In SQL, the asterisk means **all**.

The DISTINCT Keyword

In situations in which you have multiple duplicate records in a table, it might make more sense to return only unique records, instead of fetching the duplicates.
The SQL **DISTINCT** keyword is used in conjunction with SELECT to eliminate all duplicate records and return only unique ones.

The basic syntax of DISTINCT is as follows:

```
SELECT DISTINCT column_name1, column_name2
FROM table_name;
```

See the **customers** table below:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
6	Andrew	Thomas	New York
7	Daniel	Harris	New York
8	Charlotte	Walker	Chicago
9	Samuel	Clark	San Diego
10	Anthony	Young	Los Angeles

Note that there are duplicate **City** names. The following SQL statement selects only distinct values from the City column:

```
SELECT DISTINCT City FROM customers;
```

This would produce the following result. Duplicate entries have been removed.

City
New York
Los Angeles
Chicago
Houston
Philadelphia
San Diego

The LIMIT Keyword

By default, all results that satisfy the conditions specified in the SQL statement are returned. However, sometimes we need to retrieve just a subset of records. In MySQL, this is accomplished by using the **LIMIT** keyword.

The syntax for LIMIT is as follows:

```
SELECT column list
FROM table_name
LIMIT [number of records];
```

For example, we can retrieve the first **five** records from the **customers** table.

```
SELECT ID, FirstName, LastName, City
FROM customers LIMIT 5;
```

This would produce the following result:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia

Fully Qualified Names

In SQL, you can provide the table name prior to the column name, by separating them with a **dot**.
The following statements are equivalent:

```
SELECT City FROM customers;
SELECT customers.City FROM customers;
```

The term for the above-mentioned syntax is called the **"fully qualified name"** of that column.

This form of writing is especially useful when working with multiple tables that may share the same column names.

Order By

ORDER BY is used with SELECT to **sort** the returned data.

The following example sorts our **customers** table by the *FirstName* column.

```
SELECT * FROM customers
ORDER BY FirstName;
```

Result:

ID	FirstName	LastName	City
6	**Andrew**	Thomas	New York
10	**Anthony**	Young	Los Angeles
8	**Charlotte**	Walker	Chicago
3	**Chloe**	Anderson	Chicago
7	**Daniel**	Harris	New York
2	**David**	Williams	Los Angeles
4	**Emily**	Adams	Houston
5	**James**	Roberts	Philadelphia
1	**John**	Smith	New York
9	**Samuel**	Clark	San Diego

As you can see, the rows are ordered **alphabetically** by the **FirstName** column.

By default, the ORDER BY keyword sorts the results in **ascending** order.

Sorting Multiple Columns

ORDER BY can sort retrieved data by multiple columns. When using ORDER BY with more than one column, separate the list of columns to follow ORDER BY with **commas**.
Here is the **customers** table, showing the following records:

ID	FirstName	LastName	Age
1	John	Smith	35
2	David	Smith	23
3	Chloe	Anderson	27
4	Emily	Adams	34
5	James	Roberts	31
6	Andrew	Thomas	45
7	Daniel	Harris	30

To order by **LastName** and **Age**:

```
SELECT * FROM customers
ORDER BY LastName, Age;
```

This ORDER BY statement returns the following result:

ID	FirstName	LastName	Age
4	Emily	**Adams**	**34**
3	Chloe	**Anderson**	**27**
7	Daniel	**Harris**	**30**
5	James	**Roberts**	**31**
2	David	**Smith**	**23**
1	John	**Smith**	**35**
6	Andrew	**Thomas**	**45**

As we have two **Smith**s, they will be ordered by the **Age** column in ascending order.

The ORDER BY command starts ordering in the same sequence as the columns. It will order by the first column listed, then by the second, and so on.

The WHERE Statement

The **WHERE** clause is used to extract only those records that fulfill a specified criterion.

The syntax for the WHERE clause:

```
SELECT column_list
FROM table_name
WHERE condition;
```

Consider the following table:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
6	Andrew	Thomas	New York
7	Daniel	Harris	New York
8	Charlotte	Walker	Chicago
9	Samuel	Clark	San Diego
10	Anthony	Young	Los Angeles

In the above table, to SELECT a specific record:

```
SELECT * FROM customers
WHERE ID = 7;
```

ID	FirstName	LastName	City
7	Daniel	Harris	New York

SQL Operators

Comparison Operators and **Logical Operators** are used in the WHERE clause to filter the data to be selected.

The following comparison operators can be used in the WHERE clause:

Operator	Description
=	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range

For example, we can display all customers names listed in our table, with the exception of the one with ID 5.

```
SELECT * FROM customers
WHERE ID != 5;
```

Result:

ID	FirstName	LastName	City
1	John	Smith	New York
2	David	Williams	Los Angeles
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
6	Andrew	Thomas	New York
7	Daniel	Harris	New York
8	Charlotte	Walker	Chicago
9	Samuel	Clark	San Diego
10	Anthony	Young	Los Angeles

As you can see, the record with ID=5 is excluded from the list.

The BETWEEN Operator

The BETWEEN operator selects values within a range. The first value must be lower bound and the second value, the upper bound.

The syntax for the BETWEEN clause is as follows:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

The following SQL statement selects all records with IDs that fall between 3 and 7:

```
SELECT * FROM customers
WHERE ID BETWEEN 3 AND 7;
```

Result:

ID	FirstName	LastName	City
3	Chloe	Anderson	Chicago
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
6	Andrew	Thomas	New York
7	Daniel	Harris	New York

Text Values

When working with text columns, surround any text that appears in the statement with **single quotation marks (')**.

The following SQL statement selects all records in which the *City* is equal to 'New York'.

```
SELECT ID, FirstName, LastName, City
FROM customers
WHERE City = 'New York';
```

ID	FirstName	LastName	City
1	John	Smith	**New York**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**

Logical Operators

Logical operators can be used to combine two Boolean values and return a result of **true**, **false**, or **null**. The following operators can be used:

Operator	Description
AND	TRUE if **both** expressions are TRUE
OR	TRUE if **either** expression is TRUE
IN	TRUE if the operand is equal to one of a list of expressions
NOT	Returns TRUE if expression is not TRUE

When retrieving data using a SELECT statement, use logical operators in the WHERE clause to combine multiple conditions.

If you want to select rows that satisfy all of the given conditions, use the logical operator, **AND**.

ID	FirstName	LastName	Age
1	John	Smith	35
2	David	Williams	23
3	Chloe	Anderson	27
4	Emily	Adams	34
5	James	Roberts	31
6	Andrew	Thomas	45
7	Daniel	Harris	30

To find the names of the customers between 30 to 40 years of age, set up the query as seen here:

```
SELECT ID, FirstName, LastName, Age
FROM customers
WHERE Age >= 30 AND Age <= 40;
```

This results in the following output:

ID	FirstName	LastName	Age
1	John	Smith	35
4	Emily	Adams	34
5	James	Roberts	31
7	Daniel	Harris	30

You can combine as many conditions as needed to return the desired results.

OR

If you want to select rows that satisfy at least one of the given conditions, you can use the logical OR operator.

The following table describes how the logical OR operator functions:

Condition1	Condition2	Result
True	True	**True**
True	False	**True**
False	True	**True**
False	False	**False**

For example, if you want to find the customers who live either in New York or Chicago, the query would look like this:

```
SELECT * FROM customers
WHERE City = 'New York' OR City = 'Chicago';
```

Result:

ID	FirstName	LastName	City
1	John	Smith	**New York**
3	Chloe	Anderson	**Chicago**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**
8	Charlotte	Walker	**Chicago**

Combining AND & OR

The SQL **AND** and **OR** conditions may be combined to test multiple conditions in a query.

When combining these conditions, it is important to use **parentheses**, so that the order to evaluate each condition is known.

Consider the following table:

ID	FirstName	LastName	City	Age
1	John	Smith	New York	35
2	David	Williams	Los Angeles	23
3	Chloe	Anderson	Chicago	27
4	Emily	Adams	Houston	34
5	James	Roberts	Philadelphia	31
6	Andrew	Thomas	New York	45
7	Daniel	Harris	New York	30
8	Charlotte	Walker	Chicago	35
9	Samuel	Clark	San Diego	20
10	Anthony	Young	Los Angeles	33

The statement below selects all customers from the city "New York" **AND** with the age equal to "30" **OR** "35":

```
SELECT * FROM customers
WHERE City = 'New York'
AND (Age=30 OR Age=35);
```

Result:

ID	FirstName	LastName	City	Age
1	John	Smith	New York	35
7	Daniel	Harris	New York	30

You can nest as many conditions as you need.

The IN Operator

The IN operator is used when you want to compare a column with more than one value.
For example, you might need to select all customers from New York, Los Angeles, and Chicago.
With the OR condition, your SQL would look like this:

```
SELECT * FROM customers
WHERE City = 'New York'
OR City = 'Los Angeles'
OR City = 'Chicago';
```

Result:

ID	FirstName	LastName	City
1	John	Smith	**New York**
2	David	Williams	**Los Angeles**
3	Chloe	Anderson	**Chicago**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**
8	Charlotte	Walker	**Chicago**
10	Anthony	Young	**Los Angeles**

The IN Operator

You can achieve the same result with a single IN condition, instead of the multiple OR conditions:

```
SELECT * FROM customers
WHERE City IN ('New York', 'Los Angeles', 'Chicago');
```

This would also produce the same result:

ID	FirstName	LastName	City
1	John	Smith	**New York**
2	David	Williams	**Los Angeles**
3	Chloe	Anderson	**Chicago**
6	Andrew	Thomas	**New York**
7	Daniel	Harris	**New York**
8	Charlotte	Walker	**Chicago**
10	Anthony	Young	**Los Angeles**

Note the use of **parentheses** in the syntax.

The NOT IN Operator

The **NOT IN** operator allows you to exclude a list of specific values from the result set.
If we add the **NOT** keyword before **IN** in our previous query, customers living in those cities will be excluded:

```
SELECT * FROM customers
WHERE City NOT IN ('New York', 'Los Angeles', 'Chicago');
```

Result:

ID	FirstName	LastName	City
4	Emily	Adams	Houston
5	James	Roberts	Philadelphia
9	Samuel	Clark	San Diego

The CONCAT Function

The **CONCAT** function is used to concatenate two or more text values and returns the concatenating string.
Let's concatenate the *FirstName* with the *City*, separating them with a *comma*:

```
SELECT CONCAT(FirstName, ', ' , City) FROM customers;
```

The output result is:

CONCAT(FirstName, ', ' , City)
John, New York
David, Los Angeles
Chloe, Chicago
Emily, Houston
James, Philadelphia
Andrew, New York
Daniel, New York
Charlotte, Chicago
Samuel, San Diego
Anthony, Los Angeles

The AS Keyword

A concatenation results in a new column. The default column name will be the CONCAT function.
You can assign a custom name to the resulting column using the **AS** keyword:

```
SELECT CONCAT(FirstName,', ', City) AS new_column
FROM customers;
```

And when you run the query, the column name appears to be changed.

new_column
John, New York
David, Los Angeles
Chloe, Chicago
Emily, Houston
James, Philadelphia
Andrew, New York
Daniel, New York
Charlotte, Chicago
Samuel, San Diego
Anthony, Los Angeles

AVG

The **AVG** function returns the average value of a numeric column:

```
SELECT AVG(Salary) FROM employees;
```

Result:

AVG(Salary)
3100.0000

The SUM function

The **SUM** function is used to calculate the sum for a column's values.
For example, to get the sum of all of the salaries in the employees table, our SQL query would look like this:

```
SELECT SUM(Salary) FROM employees;
```

Result:

SUM(Salary)
31000

The sum of all of the employees' salaries is 31000.

The Like Operator

The **LIKE** keyword is useful when specifying a **search condition** within your WHERE clause.

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern;
```

SQL **pattern** matching enables you to use "_" to match any single character and "%" to match an arbitrary number of characters (including zero characters).
For example, to select employees whose *FirstNames* begin with the letter **A**, you would use the following query:

```
SELECT * FROM employees
WHERE FirstName LIKE 'A%';
```

Result:

ID	FirstName	LastName	Salary
6	**Andrew**	Thomas	2500
10	**Anthony**	Young	5000

As another example, the following SQL query selects all employees with a *LastName* ending with the letter "s":

```
SELECT * FROM employees
WHERE LastName LIKE '%s';
```

Result:

ID	FirstName	LastName	Salary
2	David	**Williams**	1500
4	Emily	**Adams**	4500
5	James	**Roberts**	2000
6	Andrew	**Thomas**	2500
7	Daniel	**Harris**	3000

The % wildcard can be used **multiple** times within the same pattern.

The MIN Function

The **MIN** function is used to return the minimum value of an expression in a SELECT statement.
For example, you might wish to know the minimum salary among the employees.

```
SELECT MIN(Salary) AS Salary FROM employees;
```

Salary
1500

All of the SQL functions can be combined together to create a single expression.