

PHP notes on Basic Syntax, Variables, Operators, Arrays, Control Structures (Part-1)

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
- Creating, opening, reading, writing, deleting, and closing files on the server
- Collecting form data
- Adding, deleting, and modifying information stored in your database
- controlling user-access
- encrypting data
- and much more!
- Before starting this tutorial, you should have a basic understanding of HTML. 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!

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 is easy to learn and runs efficiently on the server side.

PHP Syntax

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

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

PHP Here is an example of a simple PHP file. The PHP script uses a built in function called "echo" to output the text "Hello World!" to a web page.

```
<html>
<head>
  <title>My First PHP Page</title>
</head>
<body>
  <?php
    echo "Hello World!";
  ?>
</body>
</html>
```

- PHP statements end with semicolons (;).

Alternatively, we can include PHP in the HTML `<script>` tag.

```
<html>
<head>
  <title>My First PHP Page</title>
</head>
<body>
  <script language="php">
    echo "Hello World!";
  </script>
</body>
</html>
```

- However, the latest version of PHP removes support for `<script language="php">` tags. As such, we recommend using exclusively.

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

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

However, , as the official standard, is the recommended way of defining PHP scripts.

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 Statements

Each PHP statement must end with a semicolon

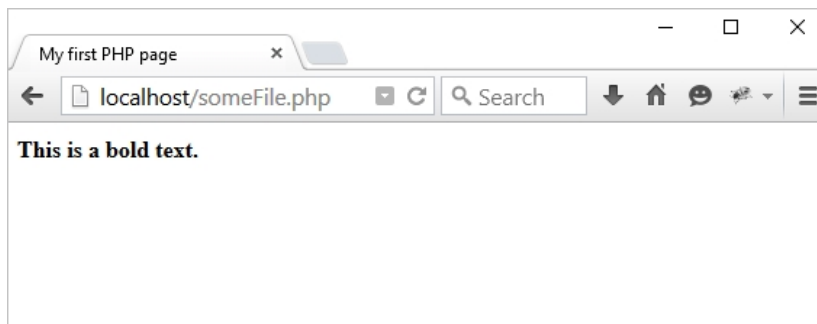
```
<?php
    echo "A";
    echo "B";
    echo "C";
?>
```

- 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>";
?>
```

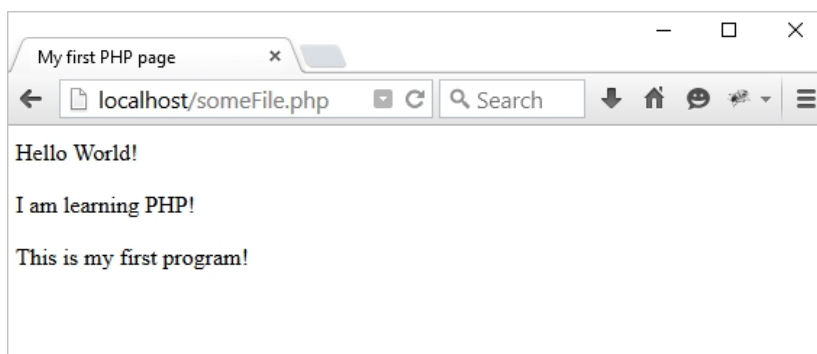


Comments

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>";
?>
```



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;
?>
```

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, case-insensitive)
```

Parameters: name: Specifies the name of the constant; value: Specifies the value of the constant; case-insensitive: Specifies whether the constant name should be case-insensitive. Default is false;

The example below creates a constant with a case-sensitive name:

```
<?php
    define("MSG", "Hi SoloLearners!");
    echo MSG;
?>
```

The example below creates a constant with a case-insensitive name:

```
<?php
    define("MSG", " Hi SoloLearners!", true);
    echo msg;
?>
```

- 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
?>
```

- Variables can store a variety of data types.

PHP Float

A **float**, or floating point number, is a number that includes a decimal point.

```
<?php
$x = 42.168;
?>
```

PHP Boolean

A Boolean represents two possible states: TRUE or FALSE.

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

- Booleans are often used in conditional testing, which will be covered later on in the course. 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();
?>
```

This script will produce an error, as the **\$name** variable has a **global scope**, and is not accessible within the **getName()** function. Tap continue to see how functions can access global variables.

- **Functions** will be discussed in the coming lessons.

The global Keyword

The global keyword is used to access a global variable from within a function. To do this, use the global keyword within the function, prior to the variables.

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

Variable Variables

With PHP, you can use one variable to specify another variable's name. So, a **variable variable** treats the value of another variable as its name.

For example:

```
<?php
    $a = 'hello';
    $hello = "Hi!";
    echo $$a;
//Outputs 'Hi!'
?>
```

- **\$\$a** is a variable that is using the value of another variable, **\$a**, as its name. The value of **\$a** is equal to "hello". The resulting variable is **\$hello**, which holds the value "Hi!".

Operators

Operators carry out operations on variables and values.

1 + 2 = 3

operand operator operand operator operand

Arithmetic Operators

Arithmetic operators work with numeric values to perform common arithmetical operations.

Operator	Name	Example
+	Addition	\$x + \$y
-	Subtraction	\$x - \$y
*	Multiplication	\$x * \$y
/	Division	\$x / \$y
%	Modulus	\$x % \$y

```
<?php
    $num1 = 8;
    $num2 = 6;

    //Addition
    echo $num1 + $num2;
    echo "<br />";

    //Subtraction
    echo $num1 - $num2;
    echo "<br />";
```

```
//Multiplication
echo $num1 * $num2;
echo "<br />";

//Division
echo $num1 / $num2;
echo "<br />";
?>
```

Modulus

The modulus operator, represented by the % sign, returns the remainder of the division of the first operand by the second operand:

```
<?php
    $x = 14;
    $y = 3;
    echo $x % $y;
?>
```

- If you use floating point numbers with the modulus operator, they will be converted to integers before the operation.

Increment & Decrement

The increment operators are used to increment a variable's value. The decrement operators are used to decrement a variable's value.

```
$x++; // equivalent to $x = $x+1;
$x--; // equivalent to $x = $x-1;
```

Increment and decrement operators either precede or follow a variable.

```
$x++; // post-increment
$x--; // post-decrement
++$x; // pre-increment
--$x; // pre-decrement
```

The difference is that the post-increment returns the original value before it changes the variable, while the pre-increment changes the variable first and then returns the value. Example:

```
$a = 2; $b = $a++; // $a=3, $b=2
$a = 2; $b = ++$a; // $a=3, $b=3
```

- The increment operators are used to increment a variable's value.

Assignment Operators

Assignment operators are used to write values to variables.

```
$num1 = 5;
$num2 = $num1;
```

\$num1 and **\$num2** now contain the value of 5.

Assignments can also be used in conjunction with arithmetic operators.

Assignment	Same as...	Description
x+=y	x = x + y	Addition
x-=y	x = x - y	Subtraction
x*=y	x = x * y	Multiplication
x/=y	x = x / y	Division
x%=y	x = x % y	Modulus

```
<?php
    $x = 50;
    $x += 100;
    echo $x;
// Outputs 150
?>
```

Comparison Operators

Comparison operators compare two values (numbers or strings). Comparison operators are used inside conditional statements, and evaluate to either TRUE or FALSE.

- Be careful using == and === ; the first one doesn't check the type of data.

Additional comparison operators:

Operator	Name	Example	Result
>	Greater than	\$x > \$y	Returns true if \$x is greater than \$y
<	Less than	\$x < \$y	Returns true if \$x is less than \$y
>=	Greater than or equal to	\$x >= \$y	Returns true if \$x is greater than or equal to \$y
<=	Less than or equal to	\$x <= \$y	Returns true if \$x is less than or equal to \$y

- The PHP comparison operators are used to compare two values (number or string).

Logical Operators

Logical operators are used to combine conditional statements.

Operator	Name	Example	Result
and	And	\$x and \$y	True if both \$x and \$y are true
or	Or	\$x or \$y	True if either \$x or \$y is true
xor	Xor	\$x xor \$y	True if either \$x or \$y is true, but not both
&&	And	\$x && \$y	True if both \$x and \$y are true
	Or	\$x \$y	True if either \$x or \$y is true
!	Not	!\$x	True if \$x is not true

- You can combine as many terms as you want. Use parentheses () for precedence.

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.

```
<?php  
    $names = array("David", "Amy", "John");  
    echo $names[1];  
    //Output 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]";  
?>
```

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.

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

Multi-Dimensional Arrays

A **multi-dimensional** array contains one or more arrays.

The dimension of an array indicates the number of indices you would need to select an element.

- For a **two-dimensional** array, you need two indices to select an element
- For a **three-dimensional** array, you need three indices to select an element Arrays more than three levels deep are difficult to manage.

Let's create a two-dimensional array that contains 3 arrays:

```
$people = array(  
    'online'=>array('David', 'Amy'),  
    'offline'=>array('John', 'Rob', 'Jack'),  
    'away'=>array('Arthur', 'Daniel')  
);
```

Now the two-dimensional \$people array contains 3 arrays, and it has two indices: row and column. To access the elements of the \$people array, we must point to the two indices.

```
<?php  
    $people = array(  
        'online'=>array('David', 'Amy'),  
        'offline'=>array('John', 'Rob', 'Jack'),  
        'away'=>array('Arthur', 'Daniel')  
    );  
  
    echo $people['online'][0]; //Outputs David
```



```
echo "<br />";
echo $people['away'][1]; //Outputs Daniel
```

?>

- The arrays in the multi-dimensional array can be both numeric and associative.
-

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.

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. The curly braces can be omitted if there only one statement after the **ifelseifelse**. **For example:**

```
if($age<=13)
echo "Child";
else
echo "Adult";
```

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:

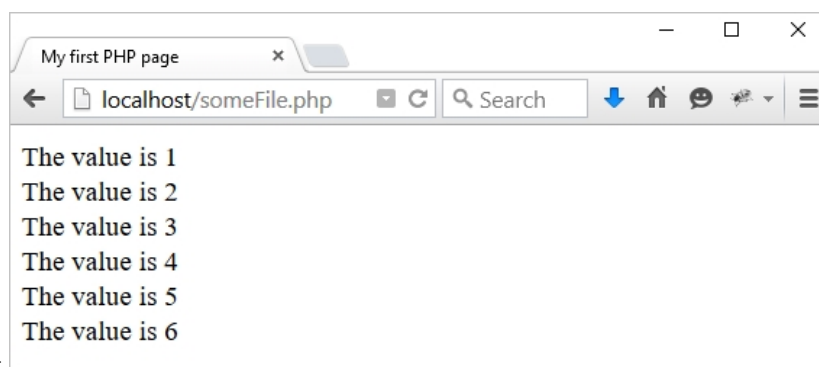
```
while (condition is true) {  
    code to be executed;  
}
```

- 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++`):

```
<?php  
$i = 1;  
while ($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 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.

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

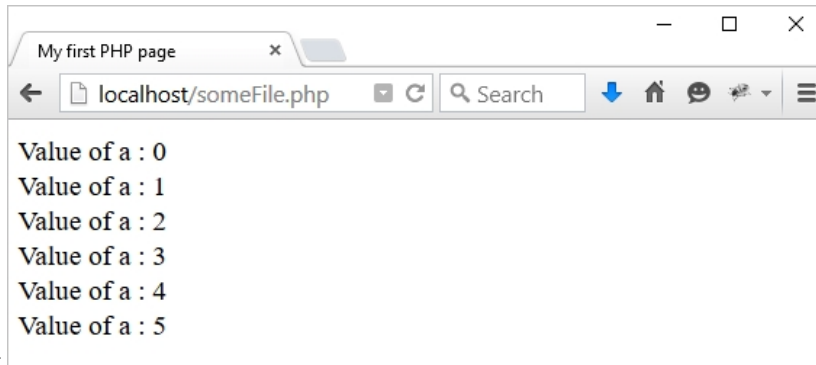
```
for (init; test; increment) {  
    code to be executed;  
}
```

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 example below displays the numbers from 0 to 5:

```
<?php
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.

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

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

```

<?php
    $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.

```

<?php
    $x=5;
    switch ($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 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.

```

<?php
    $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:

```

<?php
    $x=1;
    switch ($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:

```

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

```

- 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.

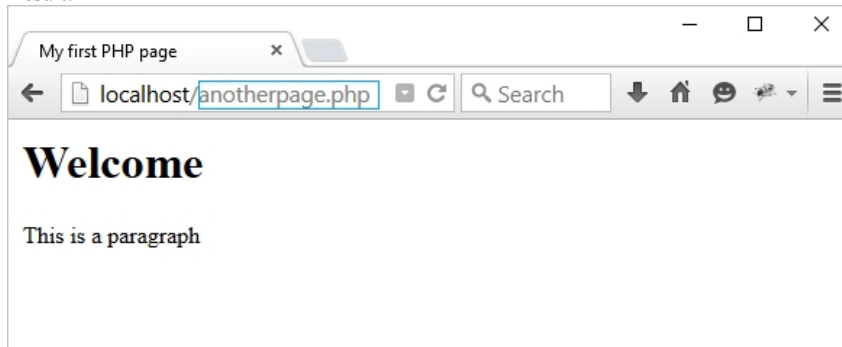
Using this approach, we have the ability to include the same header.php file into multiple pages.

```
<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.

PHP Notes on Functions, Predefined Variables, Working with Files, Object-Oriented PHP (Part-02)

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.

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:

```
<?php
    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:

```
<?php
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**

```
<?php
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**.

Default Arguments

Default arguments can be defined for the function arguments. In the example below, we're calling the function **setCounter()**. There are no arguments, so it will take on the default values that have been defined.

```
<?php
function setCounter($num=10) {
    echo "Counter is ".$num."<br />";
}
setCounter(42); //Counter is 42
setCounter(); //Counter is 10
?>
```

- When using default arguments, any defaults should be on the right side of any non-default arguments; otherwise, things will not work as expected.

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:**

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

echo mult(8, 3); // Output 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 './Playground/file0.php'
?>
```

- Our example was written in a file called **file0.php**, which is located in the root of the web server.

`$_SERVER`

`$_SERVER['HTTP_HOST']` returns the Host header from the current request.

```
<?php
echo $_SERVER['HTTP_HOST'];
//Outputs "localhost"
?>
```

This method can be useful when you have a lot of images on your server and need to transfer the website to another host. Instead of changing the path for each image, you can do the following: Create a `config.php` file, that holds the path to your images:

```
<?php
$host = $_SERVER['HTTP_HOST'];
$image_path = $host.'/images/';
?>
```

Use the `config.php` file in your scripts:

```
<?php
require 'config.php';
echo '';
?>
```

- The path to your images is now dynamic. It will change automatically, based on the Host header.
This graphic shows the main elements of `$_SERVER`.

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

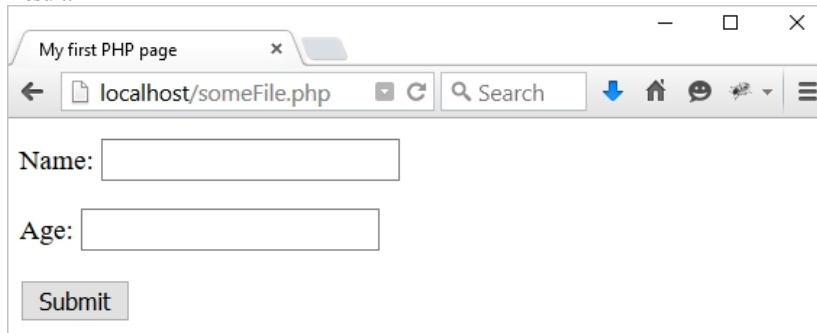
- `$_SERVER['HTTP_HOST']` returns the Host header from the current request.

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:



The screenshot shows a web browser window with the title 'My first PHP page'. The address bar shows 'localhost/someFile.php'. The page content includes a form with two text input fields. The first field is labeled 'Name:' and the second is labeled 'Age:'. Below these fields is a button labeled 'Submit'.

- The purpose of the PHP superglobals `$_GET` and `$_POST` is to collect data that has been entered into a form.

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 will be discussed in the next lesson. For now, we'll set the value to **"post"**.

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. Moreover, POST supports advanced functionality such as support for multi-part binary input while uploading files to the server. 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

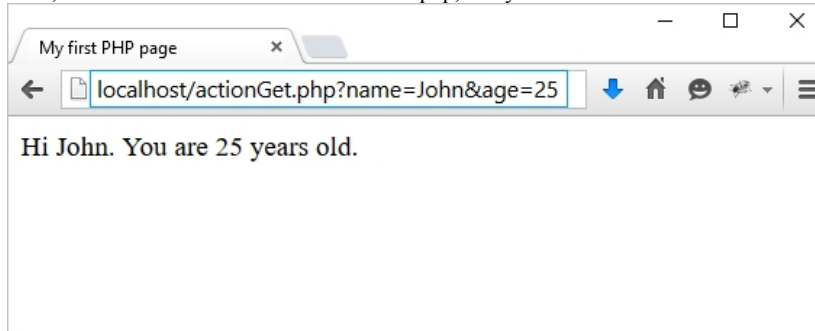
Information sent via a form using the **GET** method is visible to everyone (all variable names and values are displayed in the **URL**). **GET** also sets limits on the amount of information that can be sent - about 2000 characters. However, because the variables are displayed in the URL, it is possible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

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()**.

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, path, domain, secure, httponly);
```

~**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.

~**path**: Specifies the server path of the cookie. If set to `/`, the cookie will be available within the entire domain. If set to `/php/`, the cookie will only be available within the php directory and all sub-directories of php. The default value is the current directory in which

the cookie is being set.

~**domain**: Specifies the cookie's domain name. To make the cookie available on all subdomains of example.com, set the domain to "example.com".

~**secure**: Specifies whether or not the cookie should only be transmitted over a secure, HTTPS connection. TRUE indicates that the cookie will only be set if a secure connection exists. Default is FALSE.

~**httponly**: If set to TRUE, the cookie will be accessible only through the HTTP protocol (the cookie will not be accessible to scripting languages). Using httponly helps reduce identity theft using XSS attacks. Default is FALSE.

- The **name** parameter is the only one that's required. All of the other parameters are optional.

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. The '/' means that the cookie is available throughout the entire website.

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.

Manipulating Files

PHP offers a number of functions to use when creating, reading, uploading, and editing files. The `fopen()` function creates or opens a file. If you use `fopen()` with a file that does not exist, the file will be created, given that the file has been opened for writing (w) or appending (a).

Use one of the following modes to open the file. r: Opens file for read only. w: Opens file for write only. Erases the contents of the file or creates a new file if it doesn't exist. a: Opens file for write only. x: Creates new file for write only. r+: Opens file for read/write. w+: Opens file for read/write. Erases the contents of the file or creates a new file if it doesn't exist. a+: Opens file for read/write. Creates a new file if the file doesn't exist x+: Creates new file for read/write.

The example below creates a new file, "file.txt", which will be created in the same directory that houses the PHP code.

```
$myfile = fopen("file.txt", "w");
```

- PHP offers a number of functions to use when creating, reading, uploading, and editing files.

Write to File

When writing to a file, use the `fwrite()` function. The first parameter of `fwrite()` is the file to write to; the second parameter is the string to be written.

The example below writes a couple of names into a new file called "names.txt".

```
<?php
$myfile = fopen("names.txt", "w");

$txt = "John\n";
fwrite($myfile, $txt);
$txt = "David\n";
fwrite($myfile, $txt);

fclose($myfile);

/* File contains:
John
David
*/
?>
```

Notice that we wrote to the file "names.txt" twice, and then we used the `fclose()` function to close the file.

- The `\n` symbol is used when writing new lines.

fclose()

The `fclose()` function closes an open file and returns TRUE on success or FALSE on failure.

- It's a good practice to close all files after you have finished working with them.
-

Appending to a File

If you want to append content to a file, you need to open the file in append mode.

For example:

```
$myFile = "test.txt";
$fh = fopen($myFile, 'a');
fwrite($fh, "Some text");
fclose($fh);
```

When appending to a file using the 'a' mode, the file pointer is placed at the end of the file, ensuring that all new data is added at the end of the file.

Let's create an example of a form that adds filled-in data to a file.

```
<?php
if(isset($_POST['text'])) {
    $name = $_POST['text'];
    $handle = fopen('names.txt', 'a');
    fwrite($handle, $name."\n");
    fclose($handle);
}
?>
<form method="post">
    Name: <input type="text" name="text" />
    <input type="submit" name="submit" />
</form>
```

Now, each time a name is entered and submitted, it's added to the "names.txt" file, along with a new line.

The `isset()` function determined whether the form had been submitted, as well as whether the text contained a value.

- We did not specify an action attribute for the form, so it will submit to itself.
-

Reading a File

The `file()` function reads the entire file into an array. Each element within the array corresponds to a line in the file:

```
<?php
$myfile = fopen("names.txt", "w");

$txt = "John\n";
fwrite($myfile, $txt);
$txt = "David\n";
fwrite($myfile, $txt);

fclose($myfile);

$read = file('names.txt');
foreach ($read as $line) {
    echo $line .", ";
}
?>
```

This prints all of the lines in the file, and separates them with commas.

- We used the **foreach** loop, because the **\$read** variable is an **array**.

At the end of the output in the previous example, we would have a comma, as we print it after each element of the array. The following code lets us avoid printing that final comma.

```
<?php
$myfile = fopen("names.txt", "w");

$txt = "John\n";
fwrite($myfile, $txt);
$txt = "David\n";
fwrite($myfile, $txt);

fclose($myfile);

$read = file('names.txt');
$count = count($read);
$i = 1;
foreach ($read as $line) {
    echo $line;
    if($i < $count) {
        echo ', ';
    }
}
```

```

    }
    $i++;
}
?>

```

The `$count` variable uses the `count` function to obtain the number of elements in the `$read` array. Then, in the `foreach` loop, after each line prints, we determine whether the current line is less than the total number of lines, and print a comma if it is.

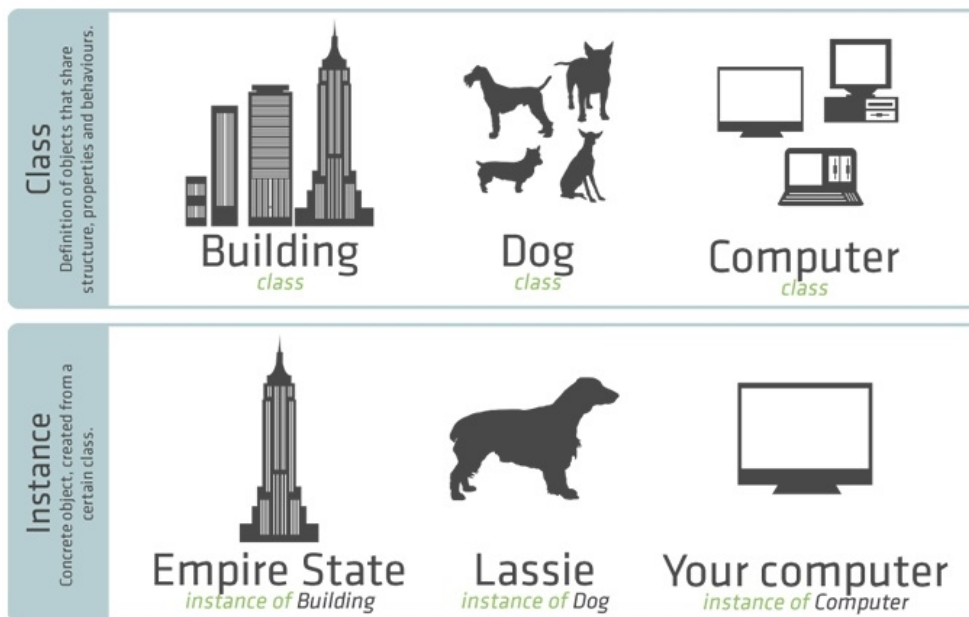
- This avoids printing that final comma, as for the last line, `$i` is equal to `$count`.

Classes & Objects in PHP

Object Oriented Programming (OOP) is a programming style that is intended to make thinking about programming closer to thinking about the real world.

Objects are created using **classes**, which are the focal point of OOP. The class describes what the object will be, but is separate from the object itself. In other words, a class can be thought of as an object's **blueprint**, description, or definition.

Take a look at the following examples:



Here, `Building` is a class. It defines the features of a generic building and how it should work. The Empire State Building is a specific object (or instance) of that class.

- You can use the same class as a blueprint for creating multiple different objects.

PHP Classes

In PHP, a class can include member variables called **properties** for defining the features of an object, and functions, called **methods**, for defining the behavior of an object. A class definition begins with the keyword **class**, followed by a class name. Curly braces enclose the definitions of the properties and methods belonging to the class.

For example:

```

class Person {
    public $age; //property
    public function speak() { //method
        echo "Hi!"
    }
}

```

The code above defines a `Person` class that includes an `age` property and a `speak()` method.

- A valid class name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. Notice the keyword **public** in front of the `speak` method; it is a **visibility specifier**. The **public** keyword specifies that the member can be accessed from anywhere in the code. There are other visibility keywords.

PHP Objects

The process of creating an object of a class is called **instantiation**. To instantiate an object of a class, use the keyword **new**, as in the example below:

```
$bob = new Person();
```

In the code above, **\$bob** is an object of the **Person** class.

To access the properties and methods of an object, use the arrow (→) construct, as in:

```
echo $bob->age;
```

This statement outputs the value of the age property for \$bob. If you want to assign a value to a property use the assignment operator = as you would with any variable.

Let's define the **Person** class, instantiate an **object**, make an assignment, and call the **speak()** method:

```
<?php
class Person {
    public $age;
    function speak() {
        echo "Hi!";
    }
}
$p1 = new Person(); // Instantiate an Object
$p1->age = 23; // Assignment
echo $p1->age; // Outputs 23
$p1->speak(); // Outputs Hi!
?>
```

\$this

\$this is a pseudo-variable that is a reference to the calling object. When working within a method, use **\$this** in the same way you would use an object name outside the class. For example:

```
<?php
class Dog {
    public $legs=4;
    public function display() {
        echo $this->legs;
    }
}
$d1 = new Dog();
$d1->display(); // 4

echo '<br />';

$d2 = new Dog();
$d2->legs = 2;
$d2->display(); // 2
?>
// Outputs:
// 4
// 2
```

We created two objects of the Dog class and called their **display()** methods. Because the **display()** method uses **\$this**, the **legs** value referred to the appropriate calling object's property value.

- As you can see, each object can have its own values for the properties of the class.
-

PHP Class Constructor

PHP provides the constructor magic method **__construct()**, which is called automatically whenever a new object is instantiated.

For example:

```
<?php
class Person {
    public function __construct() {
        echo "Object created"; // Outputs 'Object created'
    }
}
$p = new Person();
?>
```

The **__construct()** method is often used for any initialization that the object may need before it is used. Parameters can be included in **__construct()** to accept values when the object is created.

For example:

```
<?php
class Person {
    public $name;
    public $age;
    public function __construct($name, $age) {
```

```

        $this->name = $name;
        $this->age = $age;
    }
}
$p = new Person("David", 42);
echo $p->name; // Outputs 'David'
?>

```

In the code above, the constructor uses arguments in the new statement to initialize corresponding class properties.

- You can't write multiple **__construct()** methods with different numbers of parameters. Different constructor behavior must be handled with logic within a single **__construct()** method.

PHP Class Destructor

Similar to the class constructor, there is a destructor magic method **__destruct()**, which is automatically called when an object is destroyed.

For example:

```

<?php
class Person {
    public function __destruct() {
        echo "Object destroyed";
    }
}
$p = new Person();
?>

```

This script creates a new Person object. When the script ends the object is automatically destroyed, which calls the destructor and outputs the message "Object destroyed".

To explicitly trigger the destructor, you can destroy the object using the **unset()** function in a statement similar to: **unset(\$p);**

- Destructors are useful for performing certain tasks when the object finishes its lifecycle. For example, release resources, write log files, close a database connection, and so on.
- PHP releases all resources when a script finishes its execution.

```

<?php
class TestMe {
    public function __construct() { echo "2"; }
    public function __destruct() { echo "1"; }
}
$test = new TestMe();
unset($test);
?> // Outputs '21'

```

PHP Class Inheritance

Classes can inherit the methods and properties of another class. The class that inherits the methods and properties is called a **subclass**. The class a subclass inherits from is called the **parent** class.

Inheritance is achieved using the **extends** keyword.

For example:

```

<?php
class Animal {
    public $name;
    public function hi() {
        echo "Hi from Animal";
    }
}
class Dog extends Animal {
}

$d = new Dog();
$d->hi();

?> // Outputs 'Hi from Animal'

```

Here the **Dog** class inherits from the **Animal** class. As you can see, all the properties and methods of **Animal** are accessible to **Dog** objects.

Parent constructors are not called implicitly if the subclass defines a constructor. However, if the child does not define a constructor then it will be inherited from the parent class if it is not declared **private**.

- Notice all our properties and methods have **public** visibility.
- For added control over objects, declare methods and properties using a visibility keyword. This controls how and from where properties and methods can be accessed.

PHP Visibility

Visibility controls how and from where **properties** and **methods** can be accessed. So far, we have used the **public** keyword to specify that a property/method is accessible from anywhere. There are two more keywords to declare visibility:

protected: Makes members accessible only within the class itself, by inheriting, and by parent classes.

private: Makes members accessible only by the class that defines them.

Class properties must always have a visibility type. Methods declared without any explicit visibility keyword are defined as **public**.

- **Protected** members are used with inheritance.
- **Private** members are used only internally in a class.

PHP Interfaces

An interface specifies a list of methods that a class **must** implement. However, the interface itself does not contain any method implementations. This is an important aspect of interfaces because it allows a method to be handled differently in each class that uses the interface.

The **interface** keyword defines an interface. The **implements** keyword is used in a class to implement an interface.

For example, **AnimalInterface** is defined with a declaration for the **makeSound()** function, but it isn't implemented until it is used in a class:

```
<?php
interface AnimalInterface {
    public function makeSound();
}

class Dog implements AnimalInterface {
    public function makeSound() {
        echo "Woof! <br />";
    }
}

class Cat implements AnimalInterface {
    public function makeSound() {
        echo "Meow! <br />";
    }
}

$myObj1 = new Dog();
$myObj1->makeSound();

$myObj2 = new Cat();
$myObj2->makeSound();
?>
// Outputs:
// Woof!
// Meow!
```

A class can implement multiple interfaces. More than one interfaces can be specified by separating them with commas. For example:

```
class Demo implements AInterface, BInterface, CInterface {
    // Functions declared in interfaces must be defined here
}
```

An interface can be inherit another interface by using the **extends** keyword.

- All the methods specified in an interface require **public** visibility.

PHP Abstract Classes

Abstract classes can be inherited but they cannot be instantiated. They offer the advantage of being able to contain both methods with definitions and abstract methods that aren't defined until they are inherited.

A class inheriting from an abstract class must implement all the abstract methods.

The **abstract** keyword is used to create an abstract class or an abstract method.

For example:

```
<?php
abstract class Fruit {
    private $color;

    abstract public function eat();

    public function setColor($c) {
```



```

        $this->color = $c;
    }
}

class Apple extends Fruit {
    public function eat() {
        echo "Omnomnom";
    }
}

$obj = new Apple();
$obj->eat();
?>
// Outputs 'Omnomnom'

```

- Abstract functions can only appear in an abstract class.

```

<?php
abstract class Calc {
    abstract public function calculate($param);
    protected function getConst() { return 4; }
}

class FixedCalc extends Calc {
    public function calculate($param) {
        return $this->getConst() + $param;
    }
}

$obj = new FixedCalc();
echo $obj->calculate(38);
?> // Outputs '42'

```

The static Keyword

The PHP static keyword defines static properties and **static** methods. A static property/method of a class can be accessed without creating an object of that class.

A static property or method is accessed by using the **scope resolution operator ::** between the class name and the property/method name.

For example:

```

<?php
class myClass {
    static $myStaticProperty = 42;
}

echo myClass::$myStaticProperty;
?> // Outputs '42'

```

The **self** keyword is needed to access a static property from a static method in a class definition.

For example:

```

<?php
class myClass {
    static $myProperty = 42;
    static function myMethod() {
        echo self::$myProperty;
    }
}

myClass::myMethod();
?> // Outputs '42'

```

- Objects of a class cannot access static properties in the class but they can access static methods.

The final Keyword

The PHP **final** keyword defines methods that cannot be overridden in child classes. Classes that are defined final cannot be inherited.

This example demonstrates that a final method cannot be overridden in a child class:

```

<?php
class myClass {
    final function myFunction() {
        echo "Parent";
    }
}

// ERROR because a final method cannot be overridden in child classes.
class myClass2 extends myClass {
    function myFunction() {
        echo "Child";
    }
}

```

```
    }  
}  
?>  
// Outputs an Error:  
// Fatal error: Cannot override final method myClass::myFunction() in ./Playground/file0.php on line 9
```

The following code demonstrates that a final class cannot be inherited:

```
<?php  
final class myFinalClass {  
}  
  
// ERROR because a final class cannot be inherited.  
class myClass extends myFinalClass {  
}  
?>  
// Outputs an Error:  
// Fatal error: Class myClass may not inherit from final class (myFinalClass) in ./Playground/file0.php on line 7
```

- Unlike classes and methods, properties cannot be marked **final**.
-
-