**LEARN PHP**

**(with Apache server)**

### 

**Lecturer: Alexander Adu- Sarkodie**

**MSc. Telematics (IT & Telecom), MSc. Eng., Dip. Russ. Lang., Teach. Cert, AMIAEng (UK), MBCS (UK), MIfL (UK)**

**Profile -** <http://goldhawk-college.com/istudio/profile.html>

**LinkedIn -** <https://www.linkedin.com/in/alexander-adu-sarkodie-063b4b4a>

**PHP building blocks for programmes**

One of PHP’s advantages is that you can ***embed*** the PHP code directly alongside HTML. For the code to do anything it must be passed to the **PHP engine** for **interpretation** via the **web server**.

The default delimiter syntax opens with

**<?php**

and concludes with

**?>**

**echo and print statements**

The echo statement is the most **basic** statement in PHP. What this does is **output** whatever you tell it to echo (or execute). Notice when we echo a **statement** or **string**, it is **contained within quotation marks.**

<?php

echo "<div>Hi there! We are happy you have chosen this course</div>";

?>

Another way to do this is to use the print function. An example of that would be:

<?php print "I like PHP" ?>

There is a lot of debate about which is better to use or if there is any difference at all. Apparently, in very large programs that are simply outputting text the **echo** statement will run slightly faster, but for the purposes of a beginner they are interchangeable.

The ***print*** function is slightly more **dynamic** than the echo function by **returning** a value.

The *printf* function inserts dynamic variables/whatever into wherever you want with special delimiters, such as %s, or %d. For example,

<?php

printf('There is a difference between %s and %s', 'good', 'evil')

?>

would return,

'There is a difference between good and evil'.

**Alternative php syntaxes**

You can also use the short hand forms below, which requires support to be enabled on the server.

1. <? …?>
2. <?= …?> Which is equivalent to <?php echo ….;?>
3. In a pure syntactically php programme, you can leave out the closing php syntax

Use :

**<?php**

**echo phpinfo();**

**?>**

to find out list of properties, functions and characteristics of your programme installation and its environment.

**Variable declaration**

A variable **serves as storage** for holding data for a programme. Just like a storage cabinet. You can store different items in it.

A variable always **begins** with a dollar sign**, $** which is then followed by the **variable name**. Variable names can begin with either a **letter** or **an underscore** and can consist of letters, underscores, numbers or other **ASCII** characters ranging from **127** to **255**.

Variables are **case sensitive**. Once you’ve **declared** your variable you can begin to **assign values** to them. Two methodologies are available for variable assignment: by **value** and by **reference**,

By reference, you can create a **second** variable that **refers** to the **same content as another variable does**. What this means is any change to the variable referencing a particular item of variable content will be reflected among all other variables referencing that same content. You can assign variable by reference by **appending** an **ampersand (&)** to the assignment operator.

*Variable assignment:*

$color = “green”;

*Reference assignment (*The two syntaxes below are valid)

i)

<?php

$value1 = “goodbye”;

$value2 =& $value1;

$value2 = “goodbye”;

?>

ii) You can either append & after assignment operator or prepend to the old variable name.

<?php

$value1 = “goodbye”;

$value2 = &$value1;

$value2 = “goodbye”;

?>

**B. Variable scope.**

The ***location*** of the variable declaration in a program greatly **influences the realm** in which a variable can be **accessed**. PHP variables can be one of four scopes:

* **Local** variables
* **Function** parameters
* **Global** variables
* **Static** variables

*i) Local variables*

A variable declared **in a function** is considered **local** and can only **be referenced in that function**. Note that when you **exit** the function in which a local variable is declared, that **variable and its corresponding values are destroyed.**

Local variables are helpful because they **eliminate** the **possibility** **of unexpected side effect which can result from globally accessible variables that are modified intentionally or not**.

Executing this:

<?php

$x = 4;

function assignx() {

$x = 0;

printf(“\$x inside function is %d <br />” ,$x);

}

assignx();

printf(“\$x outside function is %d <br />”,$x);

?>

Results in the following,

***$x inside function is 0***

***$x outside function is 4***

*ii) Function parameters*

Function parameters are declared **after** the function name and **inside parenthesis**. Note that you can **access** and **manipulate** any function parameter **in the function in which it is declared, it is destroyed once the execution ends.**

<?php

//multiply a value by 10 and return to the caller

function x10($value) {

$finalValue = $value \* 10;

return $finalValue;

}

print x10(3);

?>

*iii) Global variables*

A **global** variable can be accessed in **any part of the program**. However, to **modify a global variable in a function, it must be explicitly declared in the function as being global. By placing the keyword GLOBAL in front of the variable**, that should be recognized as global.

In the example below,

<?php

$somevar = 15;

function addit() {

GLOBAL $somevar;

$somevar++;

}

addit();

print “somevar is $somevar”;

?>

**The displayed value of $somevar would be 16. However, if you were to omit the line**

**GLOBAL $somevar then the value will be 1 on execution.**

Another method for declaring global variables is to use PHP’s **$GLOBALS** array as below:

<?php

$somevar = 15;

function addit() {

$GLOBALS [“somevar”]++;

}

addit();

print “somevar is ” . $GLOBALS[“somevar”];

?>

*iv) Static variables*

In contrast to function parameters, which are destroyed on function exit, a **static** **variable does not lose value and will continue to hold it for further execution if the function is called again**. Static scooping is useful for **recursive functions**. Recursive functions **call themselves repeatedly until a particular condition is met.**

You can declare a variable to be static by simply placing the keyword **STATIC** in front of the variable name.

In the example below,

<?php

function keeptrack() {

STATIC $count = 0;

$count++;

echo $count;

echo “<br />”;

}

keeptrack();

keeptrack();

keeptrack()

?>

The displayed value if **not** STATIC was used would be

***1***

***1***

***1***

However, because $count **is** STATIC, **it retains its previous value each time the function is executed**. Giving the following outcome:

***1***

***2***

***3***

*v) Variable variables*

It is possible to treat at times the **content** of a variable as **dynamic**. This is achieved by placing a **second dollar sign in front of the original variable name** and again assigning another value. The program below:

<?php

$recipe = “spaghetti”;

$$recipe = “and meatballs”;

echo $recipe ${$recipe };

?>

will evaluate to : ***spaghetti and meatballs***

**C. PHP super globals**

PHP offers a number of useful **pre-defined** **variables** that are **accessible** from **anywhere within the executing script** and provides a substantial amount of **environment specific information**.

The following code will output all pre-defined variables relevant to any giving Apache web server and the scripts execution environment.

<?php

foreach ($\_SERVER as $var => $value) {

echo “$var : $value <br />”;

}

?>

You can use this code to check the user’s IP address.

<?php

printf(“your IP address is : %s” , $\_SERVER[‘REMOTE\_ADDR’]);

?>

Where %s denotes the type of data format (data type).Which is **string** in this case.

You can also gain information about a user’s browser and operating system using the code:

<?php

printf(“your browser is : %s” , $\_SERVER[‘HTTP\_USER\_AGENT’]);

?>

**D. Constants.**

A **constant** is a value that **cannot be modified** throughout the execution of the program. For example an address or a mathematical value, like “PI”. Constants are defined using the **define()** function.

The function defines a constant by **assigning a value to the name**. In the example below, the mathematical constant PI is defined and further on manipulated. Note the name is a string.

Note: Do not quote numbers if they are not intended to be strings.

<?php

define (“PI”,3.141592);

//define (“VAT”,15);

printf(“The value of PI is %f”,PI);//string name is injected as argument in this case

$pi2 = 2 \* PI;

printf(“<br />The value of PI double is %f”, $pi2 );//note the difference here:substitution.

?>

Where %f is floating point data format. One, that takes whole numbers, as well as decimals. Also, the iinput is now a variable that hosts the expression for the transition ($pi2 = 2 \* PI)

This will evaluate to:

***The value of PI is 3.141592***

***The value of PI double is 6.283184***

**E. Expressions**

An expression is a **phrase** **representing a particular action in a program**. All expressions consists of at least **one operand** and **one or more operators**. Variable declarations are expressions.

*Operands*

Operands are the **inputs** for expressions. For example, $alpha, $sum, $val1 and $val2 are all operands

<?php

$alpha;

$sum = $val1 + $val2

?>

*Operators*

An operator is a **symbol that specifies a particular action**. The **precedence** and **association** of operators are significant characteristics of any programming language. Use **BODMAS** (bracket of DIVISION, MULTIPLICATION, ADDITION, SUBTRACTION) in order of precedence in a chained expressions using multiple operators.

1. Arithmetic operators: **Perform various mathematical operations**

**Example Label Outcome**

$a + $b Addition sum of $a and $b

$a - $b Subtraction Difference of $a and $b

$a \* $b Multiplication Product of $a and $b

$a / $b Division Quotient of $a and $b

$a % $b Modulo Remainder of $a divided by $b

1. Assignment operators: These **assign a data value to a variable**.

**Example Label Outcome**

$a = 5 Assignment $a equals 5

$a += 5 Addition assignment $a equals $a plus 5

$a \*= 5 Multiplication assignment $a equals $a multiplied by 5

$a /= 5 Division Assignment $a equals $a divided by 5

$a .= 5 Concatenation assignment $a equals $a concatenated with 5

1. String operators: Provide a means for **concatenating strings** together.

There are two main operators used here. The **dot operator** (**.**) and **the concatenating assignment operator** (**.=**)

**Example Label Outcome**

$a = “abc” .”def”; Concatenation $a is assigned the string abcdef

$a .= “ghijkl”; Concatenation assignment $a equals its **current** value **concatenated** with “ghijkl”

eg;

<?php

//$a contains the string value “spaghetti and meatballs”

$a = “spaghetty” **.** ”and meatballs”;

$a .= “are delicious”;

//this will evaluate to the string :”**spaghetti and meatballs are delicious**”

?>

1. Increment and decrement operators

Increment and decrement operators provide **shortened means** by which you can **add 1 to or subtract 1 from the current value of a variable and return it.**

**Example            Label             Outcome**

++$a  $a++      Increment        Increment $a by 1 and return value

--$a  $a--         Decrement       Decrement $a by 1  and return value

1. Logical operators

Logical operators make it possible to **direct the flow of a program** and are used frequently with **control structures**, such as the **if** conditional, the **while** and **for loops**. They are also commonly used **to provide details about the outcome of other operations**. Especially, those that **return** value.

**Example Label Outcome**

$a && $b AND True if both $a and $b are true

$a AND $b AND True if both $a and $b are true

$a || $b OR True if either $a or $b is true

$a OR $b OR True if either $a or $b is true

!$a NOT True if $a is not true

NOT $a NOT True if $a is not true

$a XOR $b Exclusive OR True only if $a or $b is true but not both.

1. operators

Equality operators are used to **compare two values testing for equivalence**

**Example Label Outcome**

$a == $b Is **equal** to True if $a and $b are equivalent

$a != $b Is **not equal** to True if $a is not equal to $b

$a === $b Is **identical** to True if $a and $b are **equivalent** and the **same type.**

1. Comparison operators

Comparison operators like logical operators provide a method to direct program flow **through an examination of the comparative value of two or more variables or operands**

**Example Label Outcome**

$a < $b Less than True if $a is less than $b

$a > $b Greater than True if $a is greater than $b

$a <= $b Less than or equal to True if $a is less than or equal to $b

$a >= $b Greater than or equal to True if $a is greater than or equal to $b

($a == 12) ? 5: -1 Ternary If $a equal 12 return the value 5; otherwise return value -1

**F. String Interpolation and Escape Sequence.**

PHP offers a means for both **literal** and **figurative** interpretation. For example, the following string;

The $animal somersaulted across the lawn.\n

$animal and \n could be a literal other than PHP interpretation as a variable and new line character.

1. Escape sequence

**Sequence Description**

\n Newline character

\r Carriage return

\t Horizontal tab

\\ Backslash

\$ Dollar sign

\” Double quote

\[0-7]{1,3} Octal notation

\x[0-9A-Fa-f]{1,2} Hexadecimal notation

**G. Quotes**

i)        Double quotes

Strings enclosed in double quotes are the most commonly used in most PHP scripts. This is because they offer **more flexibility and guarantee that both variables and escapes will parse appropriately** if the right and standard syntaxes are adhered to.

The programme below

<?php

$sport = "cricket";

echo "Mike's favourite sport is $sport";

?>

will evaluate to;

**Mike's favourite sport is cricket**

Escape sequences are also parsed. **This is visible in the browser source but not in the browser window**. Use html for appropriate formatting for the browser window.

The program below;

<?php

$output = "This is one line. \n And this is another line";

echo $output ;

?>

will evaluate to in the browser

**This is one line And this is another line**

And evaluate to on two lines in source code

**This is one line**

**And this is another line**

1. Single quotes

Enclosing strings in single quotes is helpful, when the string should be interpreted **exactly** as stated. For example the program below;

<?php

print 'This string will $print exactly as it\'s \n declared';

?>

will evaluate to; **This string will $print exactly as it's \n declared**

Note it\'s is parsed or else there will be early termination. Another way is to use magic quotes. The server must be enabled for this to work.

1. Heredoc

**Heredoc** syntax offers convenient means for outputting **large amounts of text**. Rather than **delimiting with single and double quotes**.

Using this syntax,

* You create your own **identifiers**, but there are strict rules to having them parsed. The parsing of the string follows the same rules as parsing strings in double quotes. Variable and escape sequences are parsed but not the double quotes within the string.
* The **opening** and **closing** identifiers must be the same and are case sensitive

The opening should be preceded by **three angle brackets pointing to the left**

* No space should follow the opening identifier and should be on a **different line to the following string**
* The content of the string goes on the **immediate next line**
* The closing identifier must be on the **next line following the string and there should be no space before and after it.**

See example below:

<?php

$website = “http://www.romaterminal.it”;

echo <<<EXCERPT

<p>Rome's central train station, known as <a href=" $website ">Roma Terminal</a> was built in 1867 blurb

</p>

EXCERPT;

?>

This will evaluate to:

**Rome's central train station, known as Roma Terminal was built in 1867 blurb**

***Note: Rome terminal will be clickable to the site address:*** http://www.romaterminal.it;

**Working with Strings**

In some cases, it can be useful to know the exact length of a string, that is, how many characters it contains, including spaces. You may use the strlen() function for this purpose, as illustrated in this example. The function takes only one parameter, and that is the string variable to measure.

**<?php**

$string = "Hello world!";

echo "Length of string: " . strlen($string);

**?>**

**H. Lowercase and Uppercase**

Sometimes you may need to convert your string to either an uppercase or a lowercase string. Fortunately, it's very easy:

**<?php**

$string = "Hello world!";

echo strtolower($string);

echo "<br />";

echo strtoupper($string);

**?>**

The first function returns an all lowercase version of the string, while the second one returns an uppercase version. We echo out the result immediately.

**I. Finding a part of a string**

If you need to find the position of a part of a string, within a larger string, you may use the strpos() function like this:

**<?php**

$string = "Hello world!";

echo "Position of world in string: " . strpos($string, "world");

**?>**

Be aware that it's case sensitive, or in other words, "world" and "WORLD" are not the same. If you wish to look for a part of a string and case shouldn't matter, use the case-insensitive version of the function instead. It's called stripos(). We use only two parameters, but a third optional one is available, allowing you to tell PHP where to start the search. It's not relevant to this example though.

**J. Retrieving a part of a string**

In the last example, we found a part of the string. However, you may need to get an actual part of the string and use it for something else. The substr() function can help you with that, and in the next example, we use it along with two of the functions we have already covered:

**<?php**

$string = "Hello world!";

$startPos = strpos($string, "world");

$length = strlen("world");

$substring = substr($string, $startPos, $length);

echo "A specific part of the string: " . $substring;

**?>**

**K. Useful debug type functions**

* **print\_r** : Prints human readable information about a variable.
* **var\_dump :** Dumps information about a variable
* **var\_export** : Outputs or returns a parsable string representation of a variable
* **get\_defined\_vars** : Returns an array of all defined variables
* **gettype** : Get the type of a variable

**Control Structures & Handling Forms**

Control structures determine the **flow** of code within the application, defining **execution** **characteristics** such as **whether and how many times a particular code statement will execute as well as when a code block will relinquish execution**.

**L. GET, POST & Conditional Statements**

Conditional statements make it possible for your computer program to **respond accordingly** to a wide **variety of inputs**, using **logic** to **discern between various conditions based on input value**.

1. The **if** statement

The if statement is one of the most **common place constructs** of any **mainstream programming language**, offering convenient means for conditional code execution. The following is the syntax

if (expression) {

Statement to execute;

}

As an example suppose you want a congratulatory message displayed if the user guesses a **pre-determined** secret number.

<?php

$secretNumber = 045; //**First, set the result or an expected outcome and store the number**

//**Then, compare with that submitted by the user**:

if($\_POST[‘guess’] == $secretNumber) {

echo “<p>Congratulations!</p>”;

}

?>

When PHP was originally created, its main purpose was handling form data. It has later been extended to cover pretty much anything else as well, but you will soon realize that handling form data is a pretty big part of making dynamic websites. Every time you interact with the user, it's usually through a form, which is a tag containing other HTML tags which represents various form elements, e.g. a textbox, a radiobutton, a list or one of the other elements.

It's a pretty standard form. The only PHP thing we do, is to use the $\_SERVER array to get the current filename of the script and put it into the action attribute of the form tag. This will ensure that once the form is submitted, the data is sent to the exact same page. It's also possible to submit form data to another page, but in these examples, we will post it to the same page, mainly because it's nice and easy.

We specify that the method to be used when submitting the form should be POST. The other alternative is the GET method, and while there are a bunch of technical differences, you should focus on the fact that the data submitted will end up in different places depending on the method used. Also, GET data is displayed in the browser address field as parameters, while POST data is not really visible to the user.

Let's expand on the examples further above and do something with the data that is submitted through the form.

**<?php**

if(isset($\_POST[“txtName"])) {

echo "The form was submitted - your name is: " . $\_POST["txtName"];

}

**?>**

<form action="**<?php** echo $\_SERVER['PHP\_SELF']; **?>**" method="post">

Your name:

<input type="text" name="txtName" />

<input type="submit" name="btnSendForm" value="Send" />

</form>

The only real magic here is the $\_POST array. Each time a form is submitted, this array is filled with all possible values from the form being submitted. In this case, once the form is submitted, the array will contain two items, with the keys "txtName" and "btnSendForm" and their respective values. This is valuable, because it gives us easy access to the values submitted by the user. That's why we can tell the user what his or her name is, by outputting the value from the $\_POST array where "txtName" is the key.

Now, with POST forms, the data has to come from a POST request to the page. As mentioned earlier, GET forms append their data to the page URL, that is, the address you see in your browser. This of course means that you don't necessarily have to submit the data to the $\_GET array using a form. You can just as easily write the values in the address field your self. For instance, consider this example page, where we don't even have a form:

**<?php**

if(isset($\_GET[“name"])) {

echo "Your name: " . $\_GET["name"];

}

**?>**

If you call up the page in your browser, you will see nothing. However, try calling the file with a name parameter, like this:

file.php?name=John Doe

This example shows you that GET data can come directly from the user. You can try changing the first example in this chapter - simply replace post with get, and it will work just as well.

We will be looking further into FORM validation later in the course.

1. The **else** statement

The else statement is used to provide **tailored response no matter the outcome not just when it is rightly guessed.** A review of the previous example, this time offering a response in both cases is hi-lighted next.

a)

<html>

<body>

<?php

$d=**date(“D”);** // date(“D”) is a PHP pre-defined **date**  function

if($d==“Fri”){

echo “Have a nice weekend”;

echo “See you Monday”;

}

else {

echo “Have a nice day”;

}

?>

</body>

</html>

b)

<?php

$secretNumber = 453;

if($\_POST[‘guess’] == $secretNumber) {

echo “<p>Congratulations!</p>”;

}

else {

echo “<p>sorry try again</p>”;

}

?>

1. The **elseif** statement

Sometimes you will need a means for considering **each possible outcome**. This is accomplished with the *elseif* statement.

**Revising** the previous example: with the *elseif* statement. This time offering message, if the user’s guess is relatively close (say within a **range**). As seen in the quiz below.

<?php

$secretNumber = 453; //**Expected** **result**

$\_POST[‘guess’] = 461;// A **users** **guess** submitted. One of many.

//If there is a **match**

if($\_POST[‘guess’] == $secretNumber ){

echo “<p>Congratulations!</p>”;

}

//A **possible outcome** from the user for which there is **no match,**

**//**but within a **range**

elseif (($\_POST[‘guess’] - $secretNumber) <10){

echo “<p>You’re getting close</p>”;

}

//**Outside** a range

else {

echo “<p>sorry</p>”;

}

?>

1. The **switch** statement

The switch statement can be thought of as a variant of the if-else combination, often when you need to compare a **variable** against a **large number** of **values.**

Note the presence of the **break** statement at the **conclusion** of each **case** block. If a break statement is **omitted** from any loop in the program, **all subsequent case blocks will execute until a break statement is located.**

Also, notice the default statement at the end.

Ex 1

<?php

switch ($category) {

case “news”;

echo “<p>what’s happening around the world</p>”;

break;

case “weather”;

echo “<p>Your weekly forecast</p>”;

break;

case “sports”;

echo “<p>Latest sports hi-lights</p>”;

break;

default;

echo “<p>Welcome to my website</p>”;

}

?>

Ex 2

<?php

$destination = "Amsterdam";

echo "Traveling to $destination<br />";

switch ($destination){

case "Las Vegas":

echo "Bring an extra $500";

break;

case "Amsterdam":

echo "Bring an open mind";

break;

case "Egypt":

echo "Bring 15 bottles of SPF 50 Sunscreen";

break;

}?>

Ex 3

<?php

//It is always best to have a default value to use in case none of the options are found.

//Note there is no use of case in this instance.

$destination = "New York";

echo "Traveling to $destination<br />";

switch ($destination){

case "Las Vegas":

echo "Bring an extra $500";

break;

case "Amsterdam":

echo "Bring an open mind";

break;

case "Egypt":

echo "Bring 15 bottles of SPF 50 Sunscreen";

break;

default:

echo "Bring lots of underwear!";

break;

}

?>

Ex 4

<?php

//It is always best to have a default value to use in case none of the options are found.

//Note there is no use of case in this instance.

$premierClub = "FA – under construction";

switch ($premierClub){

case "Arsenal":

include('./arsenal.php');//includes Arsenal list

break;

case "Manchester U":

include('./manu.php');

break;

case "Chelsea":

include('./chelsea.php');

break;

case "Tottenham Hotspurs":

include('./spurs.php');

break;

default:

echo "No premiere club selection yet!";

break;

}

?>

1. The **while** loopstatement

The while statement specifies a **condition which must be** met before execution of its embedded code is **terminated**.

a)

<html>

<body>

<?php

$i=1;

while($i<3){

echo “The number is”. $i. “<br />”;

$i++;

}

?>

</body>

</html>

This will evaluate to:

**The number is 1**

**The number is 2**

b)

<?php

$count = 1;

while ($count <5) {

//returns formatted string with php exponential function

printf(“%d squared = %d <br />”,$count, pow($count,2));

$count++;

}

?>

The output is:

**1 squared = 1**

**2 squared = 4**

**3 squared = 9**

**4 squared = 16**

1. The **do while** statement

The do while looping statement is a variant of while but it **verifies** the loop conditional at the **conclusion of the block rather than at the beginning**. The difference between while and do while statements is that the code embedded within a while statement possibly **could never be executed,** whereas the code embedded within a *do,while* statement will **always execute at least once.**

a)

<html>

<body>

<?php

$i=1;

do{

echo “The number is” .$i. “<br />”;

$i++;

}while($i<3);

?>

</body>

</html>

This will evaluate to:

**The number is 2**

**The number is 3**

**The number is 4**

b)

<?php

$count = 11;

do {

printf (“%d squared = %d <br />”,$count ,pow($count,2));

}while($count < 10);

?>

The following is the outcome;

11 squared = 121

1. The **for** statement

The most basic *for* loops are based on a **counter**. You set the **beginning** value for the counter, set the **ending** value, and set how the counter is **incremented** or **decremented**

In the example below the variable $i can be used in a block of statements that is repeating. The following loop statement will display “Hello World” three times.

<?php

for($i=1;$i<=3;$i++) {

echo “$i. Hello World <br />”;

}

?>

The output from this statement is

1. **Hello World**
2. **Hello World**
3. **Hello World**

For loops are particularly useful to **loop** through an **array**. See below:

a)

<?php

for($i=0;$i<3;$i++) {

echo $i. “ <br />”;

}

?>

This will evaluate to:

**0**

**1**

**2**

b)

<?php

for($i=10;$i>0;$i=$i-3) {

echo $i. “ <br />”;

}

?>

This will evaluate to:

**10**

**7**

**4**

**1**

Supposing you want to know the **number of customers in the array list**, you can use the ***sizeof ()***method as part of the construct.

<?php

for($i=0;$i<sizeof($customerName);$i++) {

echo “$customerName[$i] ”; //output each member of the array

}

?>

1. The **continue statement**

The continue statement **causes the execution of the current loop iteration to** **end** and **commence at the beginning of the next iteration**. Execution of the following *while* body will recommence if the $username[$x] is found to have the value  *missing.*

<?ph

$username = array (“grace”,”doris”,”gary”,”nate”,”missing”,”tom”);

for($x=0; $x<count($username); $x++){

if($username[$x] == “missing”) {

continue;

}

printf (“Staff member: %s <br />”, $username[$x]);

}

?>

This results in the following output

**Staff member : grace**

**Staff member : doris**

**Staff member : gary**

**Staff member : nate**

**Staff member : tom**

**M.** File **inclusion** statements

Efficient programmers are always thinking in terms of ensuring **reusability** and **modularity (**Don’t Repeat Yourself**-DRY)** . The most prevalent means for ensuring such is by **isolating functional components into separate files and then re-assembling those files as needed.**

1. The **include** and **include once** statement

The **include**statement will **evaluate** and include a file into the **location** where it is called. Including a file produces the same result as **copying the data from the file specified into the location in which the statement appears.** Its prototype follows:

<?php

include(/path/to/file);

?>

For example if you want to include a series of **re-usable** features (i.e. banners and footers of a website), **pre-defined** functionsand **configuration** you could place them into a separate file (say init.php). And then include that file within the top or bottom of each php script as below.

<?php

include “/user/locallib/init.php”;

?>

You can also execute include statements **conditionally**. If an *include()* statement is placed in an ***if*** statement, the file will be included only if the *if* statement which it has been placed **evaluates to true**. Do enclose the if statements in curly braces to avoid errors.

<?php

if(expression) {

include (‘filename’)

}

else {

include(‘another filename’);

}

?>

***include\_once()*** function has the same purpose as include(), except that **it first verifies whether the file has already been included**. If a file has already been included adding the *include\_once()* **will not execute**. Otherwise it will include the file as necessary. Its prototype follows

<?php

include\_once (filename);

?>

1. The  **require** and **require once** statement

The ***require*** and ***require\_once*** statements operates in the same way as *include* and *include\_once*  **except for when it is within a conditional statement**. Using require statements will **always execute even if the conditional statements evaluates to false, so be careful!** Its prototype follows;

<?php

require (filename);

?>

The require once is useful in say **avoiding modified variables in a later inclusion of the same file.** Another problem that arises is the clashing of function names should they exist in the inclusion file.

<?php

require\_once (filename);

?>

**N. Arrays**

Arrays are **complex variables**. An array stores a **group of values** under a **single variable name**. It is useful **for storing related values**. For instance you can store information about a shirt, such as size, colour and cost in a single array named $shirt.

**Types of Array in PHP**

There are three types of array in PHP

* **Numeric** : Arrays with a ***numeric index***
* **Associative**: An array where ***each ID*** key is associated with a value. Arrays do not necessarily have to have numeric indexes for elements. The key could be arbitrarily.
* **Multi-dimensional**: An array containing ***one or more arrays.***

The simplest way of creating an array is to assign a value to a variable with square brackets [] at the end of the name. For an array named $pet with three animals, the following statement will create the array (Numeric Arrays);

<?php

$pet[1] = “dragon”;

$pet[2] = “unicorn”;

$pet[3] = “tiger”;

?>

An array can be viewed as a list of **key** **value pairs**. To get a particular value you **specify** the **key** **in the brackets**. In the preceding example the keys are 1, 2 and 3. You can also use **words** (**Associative arrays**) for keys. The following statement creates an array of country capitals;

<?php

$capitals[‘PA’] = “Paris”;

$capitals[‘LN’] = “London”;

$capitals[‘NY’] = “New York”;

?>

Another format for the above structure is:

<?php

$capitals = array(‘PA’=>’Paris’,’LN’=>’London’,’NY’=>’New York’);

?>

Note the key values are all string and quoted.

***Removing array***

To remove totally an item from the array use the **unset function** keyword in the statement

<?php

unset($pet[2]);

?>

This will remove unicorn from the array of pet.

***Sorting array***

PHP sorts arrays in a variety of ways **assigning new keys,** that are the appropriate numbers (numerical). The following statement below will sort the $pets array

<?php

sort($pets);//using the **sort()** function

?>

Consider the pet array created previously;

<?php

$pet[0] = “dragon”;

$pet[1] = “unicorn”;

$pet[2] = “tiger”;

?>

After sorting, the array becomes,

<?php

$pet[0] = “dragon”;

$pet[1] = “tiger”;

$pet[2] = “unicorn”;

?>

***Assorting arrays***

To sort arrays that use **words** for keys use the **asort()** function. This will sort by values however retaining the original key for each value

In the example below

<?php

asort($capitals);

?>

Will evaluate to;

$capitals[‘LN’] = “London”;

$capitals[‘NY’] = “New York”;

$capitals[‘PA’] = “Paris”;

Also note these:

***rsort($arrayname****)-* This will sort by value in **reverse** order and assign **new numbers** as keys

***arsort($arrayname)***– This will sort by value in reverse order and **keeps the same keys**.

***ksort($arrayname)*** – Sorts by **key**

***krsort($arrayname)*** – Sorts by **key** in **reverse** order

***usort ($arrayname, functionname)***– Sorts by **function**

***Getting values from arrays***

i) Direct Access

You can retrieve any individual value in an array by **accessing it directly** using our previous example;

<?php

$CAcapital = $capitals[‘CA’];//assign the retrieving expression to a variable

echo $CAcapital;

?>

The output from this statement is

**Sacremento**

If you include the array value in a **longer echo statement that’s enclosed by double quotes then you may have to include the array value name in curly braces like below;**

<?php

echo “The capital of England is {$capitals[‘LN’]}<br />”;

?>

**<?php**

$animals = array(2 => "Monkey", "Lion", "Turtle", "Horse");

echo $animals[2];

**?>**

The counting will then start with 2 instead. You may give all the items numbers your self, if you prefer that, as long as the keys are unique. In some cases, named keys will make more sense. No problem:

**<?php**

$namesAndAges = array("John Doe" => 45, "Jane Doe" => 33, "Dog Doe" => 11);

echo "The age of Jane Doe: " . $namesAndAges["Jane Doe"];

**?>**

We use the => operator to assign the age to the name, or in other words: The value to the key. In some cases, you may wish to add items later on, instead of when creating the array. Here is an example of that:

**<?php**

$names = array();

$names[] = "John Doe";

$names[] = "Jane Doe";

$names[] = "Dog Doe";

echo $names[1];

**?>**

We use an empty set of square brackets to make PHP create a key for the item, but obviously, you can give use the same techniques as above, when adding to the array.

ii) Using list

You can also get several values from an array using the ***list*** statement. It gets values from an array and assigns them to variables. In the example below the third value in the array is not assigned any variable and hence not copied

<?php

$shirtinfo = array (“large”,”blue”,”12.00”);

sort ($shirtinfo);

list($firstvalue.$secondvalue) = $shirtinfo;

?>

The output will be in alphabetical order

<?php

echo “$shirtinfo <br />”;

?>

and will evaluate to:

**large**

**blue**

iii) Retrieving key value pairs. For example,

In some cases you may want to retrieve keys in the array

<?php

$shirtinfo = array (“size” =>”large”, ”colour” => “blue” ,”cost” => “12.00”);

$value = $shirtinfo([‘size’])

$key = key($shirtinfo); //Note **key** is a library function

echo “$key : $value <br />”;

?>

This will evaluate to;

**size : large**

iv) Retrieving all the values of an array

You can retrieve all the values of an array with words as keys and using the **extract()** function.

<?php

extract($shirtinfo);

echo “size is $size, colour is $colour, cost is $cost”;

?>

This will evaluate to

**size is large, colour is blue, cost is 12.00**

**O. Walking through arrays using control structures**

1. **Using foreach**

Foreach walks through the array **one value at a time** and **executes** the block of statements using each value in the array. The general format is;

<?php

foreach ($arrayname as $keyname =>$valuename) {

block of statements;

}

?>

In the example that follows, the following foreach statement walks through the sample array of state capitals and echoes a list.

<?php

$capitals = array(“CA” => “Saccramento”, “TX” => “Austin”, “OR” => “Salem”);

ksort($capitals); //sort by key

// You create variables in **name value pair relationship to store each array value**. Note the **as**  keyword. Also the labels must be descriptive names that correlate with the data.

// The <br /> tag will put each record on one line.

foreach($capitals as $state => $city) {

echo “$city , $state <br />”;

}

?>

The output will be:

**Sacramento CA,**

**Salem, OR**

**Austin, TX**

The previous example can be written to output just the values,

<?php

foreach ($arrayname as $valuename) {

block of statements;

}

?>

<?php

foreach($capitals as $city) {

echo “$city <br />”;

}

?>

The webpage output will simplify to:

**Sacramento**

**Salem**

**Austin,**

Inthe next example, suppose you want to output an array of links, using the *foreach* loop statement,

<?php

$links = array(“The apache web server” => “www.apache.org”, “Apres” => “www.apres.com”,”The PHP scripting Language” => “www.php.net”);

echo “<strong>Online resources<strong>:<br />”;

foreach($links as $title => $link) {

echo “<a href=\”http://$link\”>$title</a><br />”;

}

?>

**Note**: Escape inner double quotes (**Escape sequence**).

This will result in the following:

**Online resources:<br />**

**<a href=”http:// www.apache.org”> The apache web server </a><br />**

**<a href=”** **www.apres.com”> Apres</a><br />**

**<a href=”** **www.php.net”> The PHP scripting Language </a><br />**

1. **Multidimensional arrays**

Let's walk through this more complex example. We start out by creating an empty array called contacts. We wish to store names of contact persons in it, with different names in different categories. The first array will carry a set of category names, each of them pointing to an array with the persons who belong in that specific category. We assign both category names and contact persons on the fly - each category gets a newly created array of a couple of persons attached to it.

To get a nice output of the list, we use two foreach loops - one that runs through the categories, and an inner loop which runs through the persons of the current category. We print out the name of the category, as well as the name of each person in it, followed by a linebreak. As you will see when you test this script, this creates a nice looking list of persons, divided in categories.

**<?php**

$contacts = array();

$contacts["Friends"] = array("Me", "John Doe");

$contacts["Family"] = array("Mom", "Dad");

$contacts["Enemies"] = array("Stalin", "Hitler");

foreach($contacts as $categoryName => $value) {

echo "<b>" . $categoryName . ":</b><br />";

foreach($contacts[$categoryName] as $name) {

echo $name . "<br />";

}

echo "<br />";

}

**?>**

Supposing we have the following data to process:

Shirt 20.00

Pants 22.50

Blanket 25.00

Bedspread 50.00

Lamp 44.00

Rug 75.00

We can model the above data by **aggregating** and **categorising** them into **groups**, say **clothing**, **linen** and **furniture**. This will allow **easy search** and data access than itemizing each individually as an array item. You can classify the products by putting the cost in a multi-dimensional array as follow. Since they are related by, a **common thing**. They are all **priced**.

<?php

$productPrices[‘clothing’][‘shirt’] = 20.00;

$productPrices[‘clothing’][‘pants’] = 22.50;

$productPrices[‘linens’][‘blanket’] = 25.00;

$productPrices[‘linens’][‘bedspread’] = 50.00;

$productPrices[‘furniture’][‘lamp’] = 44.00;

$productPrices[‘furniture’][‘rug’] = 75.00;

?>

The $productPrices has **three / key value pairs**. The keys are **clothing**, **linen** and **furniture**. The **value** for each **key** is an array with **two/key value pairs**. For instance, the value to the key clothing is an array with the two/key values ***shirt / 20.00*** and ***pants / 22.50***

This is a two dimensional area. The structure of this data set is illustrated below

**Product prices key value**

**Key value**

Clothing shirt 20.00

Pants 22.50

Linen blanket 30.00

Bedspread 50.00

Furniture lamp 44.00

Rug 75.00

**Extracting values from multi-dimensional arrays**

Values from multidimensional arrays can be extracted similarly to the procedure for a single dimensional arrays. Both syntaxes below can be used.

<?php

$shirtprice = $productPrices[‘clothing’][‘shirt’];

echo $shirtprice;

?>

<?php

echo $productPrices[‘clothing’][‘shirt’];

?>

However if you **combine the value within double quotes** then you will have to have **curly braces just before the dollar sign with NO SPACE!**

<?php

print “The price of a shirt is £ {$productPrices[‘clothing’][‘shirt’]}”;

?>

The output is: **The price of a shirt is £20**

**Traversing a multi-dimensional arrays**

You can walk through a multi-dimensional array using foreach statement. **One *foreach* statement for each dimensional array**. So for a two dimensional array like $productPrices you

will require **two** foreach statements. One nested in the other.

The following statements get the values from the multidimensional array and output them in an html table.

i)

<?php

$productPrices[‘clothing’][‘shirt’] = 20.00;

$productPrices[‘clothing’][‘pants’] = 22.50;

$productPrices[‘linens’][‘blanket’] = 25.00;

$productPrices[‘linens’][‘bedspread’] = 50.00;

$productPrices[‘furniture’][‘lamp’] = 44.00;

$productPrices[‘furniture’][‘rug’] = 75.00;

print “<table border=1>”;

foreach($productPrices as $productPrice) {

foreach ($productPrice as $product => $price){

$f\_price = sprintf(“%01.2f”,$price);//formats price to 2dp for 100 units

echo “<tr><td>$product : </td> <td>\$ $f\_price </td>”;”;

}

}

echo “</table>”;

?>

ii Using **count** to determine size of an array

<?php

$family = array(“Tom”,”Ian”,”Sophie”,”Alex”);

$family = array(“Pat”,”Frances”,”Athur”,”John”);

$rows = count($family);

$cols = count($family[0]);//**MUST** set first item in column field

//Column count

for($i<0;$i<$rows;$i++) {

for($j<0;$j<$cols;$j++) {

echo $family[$i][$j] . ‘ ‘;

}

echo “<br />”

}

?>

ii Using **rugged**  arrays

This is when the inner arrays are of different sizes. If this is the case you just need to check the number of columns, in each iteration.

<?php

$family = array(“Tom”,”Ian”,”Sophie”);

$family = array(“Pat”,”Frances”,”Athur”,”John”);

$rows = count($family);

for($i<0;$i<$rows;$i++) {

**$cols = count($family[$i]);//**check for position for columns before looping

for($j<0;$j<$cols;$j++) {

echo $family[$i][$j] . ‘ ‘;

}

echo “<br />”;

}

?>

## P. Implode and explode

Implode() and explode() are two very cool string functions which are related to arrays. They allow you to make an array out of a string and the other way around. Here is an example where we use both functions:

**<?php**

$values = "Rabbit|Whale|Penguin|Bird";

$array = explode("|", $values);

print\_r($array);

echo "<br /><br />";

$string = implode(" and ", $array);

echo $string;

**?>**

First of all, we define a string with a set of animals listed in it. We separate each with a |. We then use the explode function to split the values of the string into an array - each time the | character is met, the following value is appended as a new value to the array. We use the print\_r() function to make a test output of the array, to see that it actually works.

After that, we use the implode() function to join each element in the array. We have specified " and " as the first parameter, which means that each time an element is appended to the string, the " and " is appended as well. Try testing the code to see the result. Explode() and implode() can really help you out in a lot of situations.

**Is a value in the array?**

Checking whether or not a given value is in the array can quickly come in handy. Consider the following example, where we use the in\_array() function to check:

**<?php**

$animals = array("Dog", "Tiger", "Snake", "Goat");

if(in\_array("Snake", $animals))

echo "Snake is in the array!";

else

echo "No snake in the array!";

**?>**

The function can take up to three parameters, where the last one is optional. It specifies whether or not the comparison should be strict, that is, both the value and the type must match. The default parameter is false. This function checks for values. If you wish to verify whether or not a key exists in an array, use the array\_key\_exists() instead.

**Unique values**

Sometimes you will get arrays from an external source, e.g. a database or a file. If duplicate entries might occur, you may use the array\_unique() function to remove them. This function will return an array where all values are unique. For instance, try this example:

**<?php**

$array = array("Dog", "Tiger", "Snake", "Dog");

echo "Animals in array:<br />";

foreach($array as $value)

echo $value . "<br />";

echo "<br />";

$array = array\_unique($array);

echo "Animals in unique array:<br />";

foreach($array as $value)

echo $value . "<br />";

**?>**

**Getting a random element**

On many occasions you may need to get a random value from a list of options. For instance, you could have one of those "Random quotes" functions. The easiest way to do this is to gather up an array of your quotes, and then select a random entry each time. Fortunately, PHP makes it easy, with the array\_rand() function. Here is an example:

**<?php**

$animals = array("Dog", "Tiger", "Snake", "Goat");

$randomAnimal = $animals[array\_rand($animals, 1)];

echo "Random animal: " . $randomAnimal;

**?>**

The array\_rand() comes in two different flavors: If you specify 1 as the last parameter (or nothing at all, since it's the default value), a key will be returned which will give you access to the random element in your array. However, you may need more than one random element back. In that case, simply specify the amount as the second parameter. In that case, an array containing the amount of random values will be returned.

## Sorting arrays

Use the sort() function to sort a complete array, like this example:

**<?php**

$animals = array("Dog", "Tiger", "Snake", "Goat", "Rabbit", "Whale", "Bird");

echo "Unsorted animals: " . implode(", ", $animals);

echo "<br /><br />";

sort($animals);

echo "Sorted animals: " . implode(", ", $animals);

echo "<br /><br />";

$animals = array\_reverse($animals);

echo "Sorted animals, descending order: " . implode(", ", $animals);

**?>**

We simply define an array with a mixed set of animals in it. Then we output it three times. First in the original order, then sorted, using the sort() method, and then after using the array\_reverse() function, which simply turns the array around.

## Q. PHP Form Handling

The most important thing to notice when dealing with **HTML forms** and PHP is that, any form element in an HTML page will **automatically** be available to your PHP scripts. So long as the form elements have **names**.

Good because you don’t have to use extra identifiers like IDs. You can use the **names** as **keys** in the **$\_POST** or **$\_GET** arrays**.** Bad in the sense that it can invoke or trigger **Cross Site Scripting** (XSS). **Malicious code can easily be injected to any of the form fields by users and submitted**. Hence the need for sanitization and validation required.

The PHP **$\_GET** and **$\_POST** variables are used to retrieve information from forms, like user input.

### **Example**

The example below contains an HTML form with two input fields and a submit button:

<html>  
<body>  
<form action="**welcome.php**" method="post">  
 Name: <input type="text" name="**fname**" />  
 Age: <input type="text" name="**age**" />  
 <input type="submit" />  
</form>  
</body>  
</html>

When a user fills out the form above and clicks on the submit button, the form data is sent to a PHP file, called “welcome.php":

"welcome.php" looks like this:

<html>  
<body>  
Welcome <?php echo $\_POST["**fname**"]; ?>!<br />  
You are <?php echo $\_POST["**age**"]; ?> years old.  
</body>  
</html>

Output could be something like this:

Welcome John!  
You are 28 years old.

## R. Validating Form input

User input should be **validated** on the browser whenever possible (by **client scripts**). **Browser validation is faster and reduces the server load.**

You should consider server validation if the user input will be **inserted** into a **database**. A good way to **validate a form on the server is to post the form to itself, instead of jumping to a different page. The user will then get the error messages on the same page as the form**. This makes it **easier** to **discover the error**.

# PHP $\_GET Function.

The built-in **$\_GET** function is used to collect values from a form sent with **method="get".**

Information sent from a form with the **GET** method is visible to everyone (it will be displayed in the **browser's address bar**) and has limits on the amount of information to send (**max. 100 characters**).

There is one rule that you should always remember when working with data from the outside world: Never trust it! If you have a form where the user can submit various data, don't assume anything about it. Validate it before you use it! The most common security problem is the so-called Cross Site Scripting (XSS), and SQL injections, where a malicious visitor injects SQL into your database queries, allowing him to e.g. delete your tables. The most common way of doing this is either through the query string (the page address) or through a form.

Before we look into the danger of why SQL injections can be so dangerous,lets look at another danger when trusting the user data too much. Consider the following example:

**<?php**

if(isset($\_POST[“selRating"])) {

$number = $\_POST["selRating"];

echo "Selected rating: " . $number;

// Write the rating to the database here

}

**?>**

<form action="**<?php** echo $\_SERVER['PHP\_SELF']; **?>**" method="post">

Select a rating from 1 to 5:

<select name="selRating">

<option value="1">1</option>

<option value="2">2</option>

<option value="3">3</option>

<option value="4">4</option>

<option value="5">5</option>

</select>

<input type="submit" name="btnSendForm" value="Send" />

</form>

This is a minimal version of a rating system. You have probably seen them at lots of sites - a simple dropdown list lets the user choose a rating, usually from 1 to 5, and then submit the form. Behind the scenes, the rating is added to a database, and when the total rating is to be displayed, the script sums up all the ratings and then divides with the number of ratings.

It's all very simple and neat and if you run this example, you might think that this is totally safe, since the user has to select a value from the dropdown list. We use the POST method, so the user can't just enter another value in the query string. So, this is safe, right?

No, not at all. Sending POST data to a page is only a tiny bit harder than manipulating the query string of a GET request. If you have saved the above example to a file, try creating a new file, and pasting this example:

<form action="otherfile.php" method="post">

Fake rating: <input type="text" name="selRating" value="1000" />

<input type="submit" name="btnSendForm" value="Send" />

</form>

Change the action value from otherfile.php to the name you gave the file from the first example and open this one in a webbrowser. The form is a bit different, but it uses the exact same field names. If you hit the Send button, the value will be posted to otherfile.php and treated as if it came from the dropdown list - no questions asked! If you throw in a rating of 1000 instead of 5 into the database, your ratings will get completely wrong.

This is of course a small problem compared to what can be done when not validating the input. For instance, the user may submit an SQL string instead of a number, and once you try to input what you think is a number to your database, you may actually be running the malicious persons SQL code instead.

In this case, it's very easy to validate the data though. Change the PHP code in the first example to something like this:

**<?php**

if(isset($\_POST["selRating"]))

{

$number = $\_POST["selRating"];

if((is\_numeric($number)) && ($number > 0) && ($number < 6))

{

echo "Selected rating: " . $number;

// Write the rating to the database here

}

else

echo "The rating has to be a number between 1 and 5!";

}

**?>**

We have added 3 simple checks:

* First of all, it has to be a number, which we check with the is\_numeric() function. It simply tells us if the parameter we pass is a number or at least a string that can be converted directly to a number.
* After that, we require that $number has to be bigger than 0, and smaller than 6, which leaves us with 1-5 as the only possible values.
* We have just secured the form. Every time you use a piece of data which comes from the user, you should think about how to validate it properly before using it.

### **Example**

<form action="welcome.php" **method="get"**>  
 Name: <input type="text" name="**fname**" />  
 Age: <input type="text" name="**age**" />  
 <input type="submit" />  
</form>

When the user clicks the "Submit" button, the URL sent to the server could look something like this:

<http://www.learnphp/welcome.php?fname=Peter&age=37>

The "welcome.php" file can now use the $\_GET function to collect form data (**the names of the form fields will automatically be the keys in the $\_GET array**):

Welcome <?php echo $\_GET["fname"]; ?>.<br /> //**fname** and **age** are **keys** in the $\_GET array   
You are <?php echo $\_GET["age"]; ?> years old!

## When to use method="get"?

When using method="get" in HTML forms, all variable names and values are displayed in the URL.

**Note:** This method **should not be used when sending passwords or other sensitive information**!

However, because the variables are displayed in the URL, it is possible to **bookmark** the page. This can be useful in some cases like quiz applications or lesser high risk applications. For search engine optimisation (SEO).

**Note:** The get method is not suitable for large variable values; the **value cannot exceed 100 characters in most browsers.**

# PHP $\_POST Function

## The $\_POST Function

The built-in **$\_POST** function is used to collect values from a form sent with **method="post".**

Information sent from a form with the POST method is **invisible** to others and has **no limits** on **the amount of information to send.**

**Note:** However, there is an **8 Mb max size** for the POST method, by **default** (can be changed by setting the **post\_max\_size in the php.ini file**).

**Example**

<form action="welcome.php" **method="post">**  
 Name: <input type="text" name="**fname**" />  
 Age: <input type="text" name="**age**" />  
 <input type="submit" />  
</form>

When the user clicks the "Submit" button, the URL will look like this:

<http://www.learnphp/welcome.php>

The "welcome.php" file can now use the $\_POST function to collect form data (the **names** of the form fields will automatically be the **keys** in the **$\_POST array**):

Welcome <?php echo $\_POST["**fname**"]; ?>!<br />  
You are <?php echo $\_POST["**age**"]; ?> years old.

## When to use method="post"?

Information sent from a form with the POST method is **invisible** to others and has no **limits** on the amount of information to send (Default value is 8MB, but this can be changed in the .ini file).

However, because the variables are not displayed in the URL, it is not **possible to bookmark** the page. For say SEO purposes.

## The PHP $\_REQUEST Function

The PHP built-in **$\_REQUEST** function contains the contents of both **$\_GET**, **$\_POST**, and **$\_COOKIE**.

The $\_REQUEST function can be used to collect form data sent with both the GET and POST methods.

### **Example**

Welcome <?php echo **$\_REQUEST**["fname"]; ?>!<br />  
You are <?php echo **$\_REQUEST**["age"]; ?> years old.

# S. PHP Sending E-mails

PHP allows you to send e-mails directly from a script.

## The PHP mail() Function

The PHP mail() function is used to send emails from inside a script.

**Syntax**

**mail(to,subject,message,headers,parameters)**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| To | Required. Specifies the receiver / receivers of the email |
| subject | Required. Specifies the subject of the email. **Note:** This parameter cannot contain **any newline characters** |
| message | Required. Defines the message to be sent. Each line should be separated with a LF **(\n).** Lines should not exceed **70** characters |
| headers | Optional. Specifies additional headers, like From, Cc, and Bcc. The additional headers should be separated with a CRLF (\r\n) |
| parameters | Optional. Specifies an additional parameter to the sendmail program |

## PHP Simple E-Mail

The simplest way to send an email with PHP is to send a text email.

In the example below we first declare the variables ($to, $subject, $message, $from, $headers), then we use the variables in the **mail()** function to send an e-mail:

<?php  
 $to = "someone@example.com";  
 $subject = "Test mail";  
 $message = "Hello! This is a simple email message.";  
 $from = "someonelse@example.com";  
 $headers = "From: $from";  
 mail($to,$subject,$message,$headers);  
 echo "Mail Sent.";  
?>

## PHP Mail Form

With PHP, you can create a feedback-form on your website. The example below sends a text message to a specified e-mail address:

<html>  
<body>  
<?php  
if (isset($\_REQUEST['email'])) {   
   //if "email" is filled out, send email  
   //send email  
   $email = $\_REQUEST['email'] ;  
   $subject = $\_REQUEST['subject'] ;  
   $message = $\_REQUEST['message'] ;  
   mail( "someone@example.com", "Subject: $subject",  
   $message, "From: $email" );  
   echo "Thank you for using our mail form";  
  }

else   {  
 //if "email" is not filled out, display the form  
   echo "<form method='post' action='mailform.php'>  
   Email: <input name='email' type='text' /><br />  
   Subject: <input name='subject' type='text' /><br />  
   Message:<br />  
   <textarea name='message' rows='15' cols='40'></textarea><br />  
   <input type='submit' />  
   </form>";  
  }  
?>  
</body>  
</html>

This is how the example above works:

* First, check if the email input field is filled out
* If it is not set (like when the page is first visited); output the HTML form
* If it is set (after the form is filled out); send the email from the form
* **When submit is pressed after the form is filled out, the page reloads, sees that the email input is set, and sends the email.**

The problem with the code above is that unauthorized users can insert data into the **mail headers** via the input form and send or redirect the mail.

## T. PHP Stopping E-mail Injections

The best way to stop **e-mail injections** is to **validate** the **input**.

The code below is the same as in the previous chapter, but now we have added an input validator that checks the email field in the form:

In the code above we use **PHP filters** to validate input:

* The **FILTER\_SANITIZE\_EMAIL** filter removes all **illegal e-mail characters** from a string
* The **FILTER\_VALIDATE\_EMAIL** filter validates value as an e-mail address

<html>

<body>

<?php

function spamcheck($field){

//filter\_var() sanitizes the e-mail

//address using FILTER\_SANITIZE\_EMAIL

$field=filter\_var($field, FILTER\_SANITIZE\_EMAIL);

//filter\_var() validates the e-mail

//address using FILTER\_VALIDATE\_EMAIL

if(filter\_var($field, FILTER\_VALIDATE\_EMAIL)){

return TRUE;

}

else {

return FALSE;

}

}

if (isset($\_REQUEST['email'])){

//if "email" is filled out, proceed

//check if the email address is invalid

$mailcheck = spamcheck($\_REQUEST['email']);

if ($mailcheck==FALSE) {

echo "Invalid input";

}

else {

//send email

$email = $\_REQUEST['email'] ;

$subject = $\_REQUEST['subject'] ;

$message = $\_REQUEST['message'] ;

mail("someone@example.com", "Subject: $subject",

$message, "From: $email" );

echo "Thank you for using our mail form";

}

}

else{

//if "email" is not filled out, display the form

echo "<form method='post' action='mailform.php'>

Email: <input name='email' type='text' /><br />

Subject: <input name='subject' type='text' /><br />

Message:<br />

<textarea name='message' rows='15' cols='40'></textarea><br />

<input type='submit' />

</form>";

}

?>

</body>

</html>

## A. Introduction to PHP classes

After the introduction to classes in the previous chapter, we're now ready to write our very own class. It will hold information about a generic user, for instance a user of your website.

A class definition in PHP looks pretty much like a function declaration, but instead of using the function keyword, the class keyword is used. Let's start with a stub for our User class:

<?php

class User {

}

?>

This is as simple as it gets, and as you can probably imagine, this class can do absolutely nothing at this point. We can still instantiate it though, which is done using the new keyword:

$user = new User();

But since the class can't do anything yet, the $user object is just as useless. Let's remedy that by adding a couple of class variables and a method:

class User {

public $name;

public $age;

public function Describe()

{

return $this->name . " is " . $this->age . " years old";

}

}

Okay, there are a couple of new concepts here. First of all, we declare two class variables, a name and an age. The variable name is prefixed by the access modifier "public", which basically means that the variable can be accessed from outside the class. We will have much more about access modifiers in one of the next chapters.

Next, we define the Describe() function. As you can see, it looks just like a regular function declaration, but with a couple of exceptions. It has the public keyword in front of it, to specify the access modifier. Inside the function, we use the "$this" variable, to access the variables of the class it self. $this is a special variable in PHP, which is available within class functions and always refers to the object from which it is used.

Now, let's try using our new class. The following code should go after the class has been declared or included:

$user = new User();

$user->name = "John Doe";

$user->age = 42;

echo $user->Describe();

The first thing you should notice is the use of the -> operator. We used it in the Describe() method as well, and it simply denotes that we wish to access something from the object used before the operator. $user->name is the same as saying "Give me the name variable on the $user object". After that, it's just like assigning a value to a normal variable, which we do twice, for the name and the age of the user object. In the last line, we call the Describe() method on the user object, which will return a string of information, which we then echo out. The result should look something like this:

*John Doe is 42 years old*

## B. Constructors and destructors

A constructor and a destructor are special functions which are automatically called when an object is created and destroyed. The constructor is the most useful of the two, especially because it allows you to send parameters along when creating a new object, which can then be used to initialize variables on the object. Here's an example of a class with a simple constructor:

class Animal {

public $name = "No-name animal";

public function \_\_construct()

{

echo "I'm alive!";

}

}

As you can see, the constructor looks just like a regular function, except for the fact that it starts with two underscores. In PHP, functions with two underscore characters before the name usually tells you that it's a so-called magic function, a function with a specific purpose and extra functionality, in comparison to normal functions. So, a function with the exact name "\_\_construct", is the constructor function of the class and will be called automatically when the object is created. Let's try doing just that:

$animal = new Animal();

With just that line of code, the object will be created, the constructor called and the lines of code in it executed, which will cause our "I'm alive!" line to be outputted. However, as mentioned previously, a big advantage of the constructor is the ability to pass parameters which can be used to initialize member variables. Let's try doing just that:

<?php

class Animal {

public $name = "No-name animal";

public function \_\_construct($name) {

$this->name = $name;

}

}

$animal = new Animal("Bob the Dog");

echo $animal->name;

?>

Declaring the constructor with parameters is just like declaring a regular function, and passing the parameter(s) is much like calling a regular function, but of course with the "new" keyword that we introduced earlier. A constructor can have as many parameters as you want.

## C. Destructors

A destructor is called when the object is destroyed. In some programming languages, you have to manually dispose of objects you created, but in PHP, it's handled by the Garbage Collector, which keeps an eye on your objects and automatically destroys them when they are no longer needed. Have a look at the following example, which is an extended version of our previous example:

<?php

class Animal {

public $name = "No-name animal";

public function \_\_construct($name) {

echo "I'm alive!";

$this->name = $name;

}

public function \_\_destruct()

{

echo "I'm dead now :(";

}

}

$animal = new Animal("Bob");

echo "Name of the animal: " . $animal->name;

?>

As you can see, the destructor is just like a constructor, only the name differs. If you try running this example, you will see first the constructor message, then the name of the animal that we manually output in the last line, and after that, the script ends, the object is destroyed, the destructor is called and the message about our poor animal being dead is outputted.

# D. Visibility (Scope Descriptors)

Visibility is a big part of OOP. It allows you to control where your class members can be accessed from, for instance to prevent a certain variable to be modified from outside the class. The default visibility is public, which means that the class members can be accessed from anywhere. This means that declaring the visibility is optional, since it will just fall back to public if there is no access modifier.

For backwards compatibility, the old way of declaring a class variable, where you would prefix the variable name with the "var" keyword (this is from PHP 4 and should not be used anymore) will also default to public visibility.

PHP is pretty simple in this area, because it comes with only 3 different access modifiers: private, protected and public.

**Private members** can only be accessed from inside the class itself.

**Protected members** can only be accessed from inside the class it self and its child classes.

**Public members** can be accessed from anywhere - outside the class, inside the class it self and from child classes.

# E. Inheritance

Inheritance is one of the most important aspects of OOP. It allows a class to inherit members from another class. Understanding why this is smart without an example can be pretty difficult, so let's start with one of those.

Imagine that you need to represent various types of animals. You could create a Cat class, a Dog class and so on, but you would probably soon realize that these classes would share quite a bit of functionality. On the other hand, there could be stuff that would have to be specific for each animal. For a case like this, inheritance is really great. The idea is to create a base class, in this case called Animal, and then create a child class for each specific animal you need. Another advantage to this approach is that you will every animal you have will come with the same basic functionality that you can always rely on.

Again, this can seem very theoretic and you might not find it very useful in the beginning, but as you create more advanced websites, you will likely run into situations where inheritance can come in handy. Let's have a look at an example now:

class Animal {

public $name;

public function Greet() {

return "Hello, I'm some sort of animal and my name is " . $this->name;

}

}

A very simple class, pretty much like the ones we created in a previous chapter. However, "some sort of animal" is not very descriptive, so let's create a child class for a dog:

class Dog extends Animal {

}

The dog is declared like a regular class, but after that, we use the extends keyword to tell PHP that the Dog class should inherit from the Animal class. Right now, our Dog class has the exact same functionality as the Animal class. Verify this by running the following code:

$dog = new Dog();

echo $dog->Greet();

You will see that both the name and the Greet() function is still there, but they are also still very anonymous. Let's change that by writing a specific version of the Greet() function for our Dog:

class Dog extends Animal {

public function Greet() {

return "Hello, I'm a dog and my name is " . $this->name;

}

}

Notice that we declare the Greet() function again, because we need for it to do something else, but the $name class variable is not declared - we already have that defined on the Animal class, which is just fine.

As you can see, even though $name is not declared on the Dog class, we can still use it in its Greet() function. Now, with both classes declared, it's time to test them out. The following code will do that for us:

$animal = new Animal();

echo $animal->Greet();

$animal = new Dog();

$animal->name = "Bob";

echo $animal->Greet();

We start out by creating an instance of an Animal class and then call the Greet() function. The result should be the generic greeting we wrote first. After that, we assign a new instance of the Dog class to the $animal variable, assign a real name to our dog and then call the Greet() function again. This time, the Dog specific Greet() function is used and we get a more specific greeting from our animal, because it's now a dog.

Inheritance works recursively as well - you can create a class that inherits from the Dog class, which in turn inherits from the Animal class, for instance a Puppy class. The Puppy class will then have variables and methods from both the Dog and the Animal class.

## F. Abstract classes

Abstract classes are special because they can never be instantiated. Instead, you typically inherit a set of base functionality from them in a new class. For that reason, they are commonly used as the base classes in a larger class hierarchy. In the chapter on inheritance, we created an Animal class and then a Dog class to inherit from the Animal class.

In your project, you may very well decide that no one should be able to instantiate the Animal class, because it's too unspecific, but instead use a specific class inheriting from it. The Animal class will then serve as a base class for our own little collection of animals.

A method can be marked as abstract as well. As soon as you mark a class function as abstract, you have to define the class as abstract as well - only abstract classes can hold abstract functions. Another consequence is that you don't have to (and can't) write any code for the function - it's a declaration only.

You would do this to force anyone inheriting from your abstract class to implement this function and write the proper code for it. If you don't, PHP will throw an error. However, abstract classes can also contain non-abstract methods, which allows you to implement basic functionality in the abstract class. Let's go on with an example. Here is the abstract class.

abstract class Animal {

public $name;

public $age;

public function Describe() {

return $this->name . ", " . $this->age . " years old";

}

abstract public function Greet();

}

As you can see, it looks like a regular exception, but with a couple of differences. The first one is the abstract keyword, which is used to mark both the class it self and the last function as abstract. As mentioned, an abstract function can't contain any body (code), so it's simply ended with a semi-colon as you can see. Now let's create a class that can inherit our Animal class:

class Dog extends Animal {

public function Greet() {

return "Woof!";

}

public function Describe() {

return parent::Describe() . ", and I'm a dog!";

}

}

As you can see, we implement the both functions from the Animal class. The Greet() function we are forced to implement, since it's marked as abstract - it simply returns a word/sound common to the type of animal we are creating. We are not forced to implement the Describe() function - it's already implemented on the Animal class, but we would like to extend the functionality of it a bit.

Now, the most helpful part is that we can re-use the code implemented in the Animal class, and then add to it as we please. In this case, we use the parent keyword to reference the Animal class, and then we call Describe() function on it. We then add some extra text to the result, to clarify which type of animal we're dealing with. Now, let's try using this new class:

$animal = new Dog();

$animal->name = "Bob";

$animal->age = 7;

echo $animal->Describe();

echo $animal->Greet();

Nothing fancy here, really. We just instantiate the Dog class, set the two properties and then call the two methods defined on it. If you test this code, you will see that the Describe() method is now a combination of the Animal and the Dog version, as expected.

## G. Static classes

Since a class can be instantiated more than once, it means that the values it holds, are unique to the instance/object and not the class itself. This also means that you can't use methods or variables on a class without instantiating it first, but there is an exception to this rule.

Both variables and methods on a class can be declared as static (also referred to as "shared" in some programming languages), which means that they can be used without instantiating the class first. Since this means that a class variable can be accessed without a specific instance, it also means that there will only be one version of this variable.

Another consequence is that a static method cannot access non-static variables and methods, since these require an instance of the class.

In a previous exercise, we wrote a User class. Let's expand it with some static functionality, to see what the fuzz is all about:

**<?php**

class User {

public $name;

public $age;

public static $minimumPasswordLength = 6;

public function Describe() {

return $this->name . " is " . $this->age . " years old";

}

public static function ValidatePassword($password) {

if(strlen($password) >= self::$minimumPasswordLength)

return true;

else

return false;

}

}

$password = “test";

if(User::ValidatePassword($password))

echo "Password is valid!";

else

echo "Password is NOT valid!";

**?>**

What we have done to the class, is adding a single static variable, the $minimumPasswordLength which we set to 6, and then we have added a static function to validate whether a given password is valid. I admit that the validation being performed here is very limited, but obviously it can be expanded.

Now, couldn't we just do this as a regular variable and function on the class? Sure we could, but it simply makes more sense to do this statically, since we don't use information specific to one user - the functionality is general, so there's no need to have to instantiate the class to use it.

As you can see, to access our static variable from our static method, we prefix it with the self keyword, which is like this but for accessing static members and constants. Obviously it only works inside the class, so to call the ValidatePassword() function from outside the class, we use the name of the class. You will also notice that accessing static members require the double-colon operator instead of the -> operator, but other than that, it's basically the same.

## H. Class constants

A constant is, just like the name implies, a variable that can never be changed. When you declare a constant, you assign a value to it, and after that, the value will never change. Normally, simple variables are just easier to use, but in certain cases constants are preferable, for instance to signal to other programmers (or your self, in case you forget) that this specific value should not be changed during runtime.

Class constants are just like regular constants, except for the fact that they are declared on a class and therefore also accessed through this specific class. Just like with static members, you use the double-colon operator to access a class constant. Here is a basic example:

**<?php**

class User {

const DefaultUsername = "John Doe";

const MinimumPasswordLength = 6;

}

echo "The default username is " . User::DefaultUsername;

echo "The minimum password length is " . User::MinimumPasswordLength;

**?>**

As you can see, it's much like declaring variables, except there is no access modifier - a constant is always publicly available. As required, we immediately assign a value to the constants, which will then stay the same all through execution of the script. To use the constant, we write the name of the class, followed by the double-colon operator and then the name of the constant.

## I. The "final" keyword

Previously, we saw how we could let a class inherit from another class. We also saw how you could override a function in an inherited class, to replace the behaviour originally provided. However, in some cases you may want to prevent a class from being inherited from or a function to be overridden. This can be done with the final keyword, which simply causes PHP to throw an error if anyone tries to extend your final class or override your final function.

A final class could look like this:

final class Animal {

public $name;

}

A class with a final function could look like this:

class Animal {

final public function Greet() {

return "The final word!";

}

}

The two can be combined if you need to, but they can also be used independently, as seen in the examples above.

## A. Connecting MySQL with PHP

A database is your best choice for storing data in your web application, and the MySQL database server has always been the most popular choice among PHP developers. It's supported by almost any hosting company offering PHP, which makes it easy to get started with, and you can even download and install it on your own computer, for testing purposes.

MySQL uses the SQL (Structured Query Language) programming language to work with the data, and PHP interacts with MySQL by simply passing SQL code through a set of MySQL functions to the MySQL server, which then returns a result that PHP can interpret. It can seem a bit scary to have to learn a second language to interact with databases, but fortunately SQL is a fairly simple language, which looks a lot like the English language and we will provide you with some good SQL examples, allowing you to do the most common tasks.

In the following chapters we will work with the MySQL database and make it do various things for us. To do it properly, we need some common test data, which you will need to add to a database for which you have access to. The easiest way to do this is to use one of the many MySQL tools, with the most popular one being phpMyAdmin, which is installed on most servers offering PHP and MySQL.

CREATE TABLE `test\_users` (

`id` int(**11**) NOT NULL auto\_increment,

`name` varchar(**100**) NOT NULL default '',

`country` varchar(**100**) NOT NULL default '',

PRIMARY KEY (`id`)

);

INSERT INTO `test\_users` VALUES (**1**,'David','USA');

INSERT INTO `test\_users` VALUES (**2**,'Sammy','Canada');

INSERT INTO `test\_users` VALUES (**3**,'Heidi','Germany');

INSERT INTO `test\_users` VALUES (**4**,'Pierre','France');

INSERT INTO `test\_users` VALUES (**5**,’Carlos','Spain');

When done, a new table called "test\_users" should have been created and filled with a small amount of testing data.

## B. Establishing a connection

We have managed to create a database table with some test data. Now we need to establish a connection to the database server so that we can start working with the data. Connecting to a MySQL database with PHP is very easy. It can be done using a single function from the nice array of MySQL related functions in PHP:

mysql\_connect("localhost", "username", "password");

If default values have been configured in your configuration file, you can leave out these parameters, but otherwise you will need to specify a host and typically a username and a password for it. Both the username and password has been set by yourself or by your hosting company. If you host MySQL on your own machine, or if you run your PHP code on your hosting company's server, you can usually just specify localhost as the host value. If in doubt, ask your hosting company or check their support pages.

The mysql\_connect() function returns a resource, a direct link to the database server, which should be used to access the database server each time you use one of the MySQL functions. However, if this resource is not specified, PHP will just use the last opened connection, allowing you to write less code. In most cases you will only need one MySQL connection per page, so this should work just fine for you.

To work with a database, you need to call one more function, the mysql\_select\_db() function. The name really tells it all - for a specific connection, it selects a database that you will be working with. It's very simple to use:

mysql\_select\_db("my\_database");

This should be called after you have established the connection using mysql\_connect(). It takes one or two parameters. The first should be the name of the database you wish to use. The second one is optional and allows you to specify which MySQL resource link the function should be performed on. Here is an example where we use these two essential functions together:

mysql\_connect("localhost", "username", "password");

mysql\_select\_db(“my\_database");

And here is the same example, but where we get the resource link from mysql\_connect() and use it. This is the way you should do it mainly if you need to connect to more than one database server on the same page:

$dbConnection = mysql\_connect("localhost", "username", "password");

mysql\_select\_db("my\_database", $dbConnection);

With this, we now have a connection to the database server and we have selected the desired database. In the next chapter, we will use this connection, but before we do so, we should talk a bit about closing the connection again. A connection to a database server is costly, so it should obviously be closed once we're done using it.

However, PHP can and will do this for us automatically, if we choose not to do it, once the page is fully executed. If we for some reason want to close a MySQL connection before the page is done executing, we can do it by using the mysql\_close() function:

mysql\_close();

This will close the last opened connection. If we want to close a specific connection, just pass its link to the function:

mysql\_close($dbConnection);

## C. Retrieving data

We established a connection to your MySQL database, so by now its time to do something interestingly with the connection. As mentioned in the introduction, we interact with the MySQL server by sending SQL code through a PHP function, more specifically the mysql\_query() function.

To retrieve data through SQL, the SELECT statement is used. It comes in many variations, but it its most simple form, it typically looks like this:

SELECT column\_name FROM table\_name

This query will select data from the column called "column\_name" in the table called "table\_name" and return all rows of it. In the introduction chapter, we inserted some rows of data in our test\_users table, where we have the columns id, name and country. To get this data, our query could look like this:

SELECT id, name, country FROM test\_users

That's the SQL needed to retrieve our data. Let's make PHP execute it for us, by using the mysql\_query() function:

mysql\_connect("localhost", "username", "password");

mysql\_select\_db("my\_database");

$query = mysql\_query("SELECT id, name, country FROM test\_users”);

The mysql\_query() function simply sends the SQL code to the MySQL server and then returns a link to the result. We will then have to use one of the many related PHP functions to actually use the result. For instance, we can see how many rows were returned by using the mysql\_num\_rows() function (the connection should already be made, as shown above):

$query = mysql\_query("SELECT id, name, country FROM test\_users");

echo "The table currently contains " . mysql\_num\_rows($query) . " row(s)”;

Of course, if we only wanted the number of rows, there would be more efficient ways of doing it and there would be no reason for selecting all three rows. Instead of just getting the number of rows, let's try actually getting some data out:

$query = mysql\_query("SELECT id, name, country FROM test\_users");

echo "The first name is: " . mysql\_result($query, 0, “name");

The mysql\_result() can pick out a single piece of data from a result link. As parameters, we specify the query link, the row index and the name of the column we want data for, so in this case we get a result from the $query result reference, we use the first row of returned (row number zero) and we get data from the column called "name". That will get us the first name in the table, which is then outputted.

In most situations you might need all the rows of data that you select, along with all the columns you select. This is usually done with the mysql\_fetch\_array() function, which simply gets you an entire row of data as an array, while moving the internal pointer one step ahead, so that the function will get you the row after that the next time you call it. This makes it excellent for using with a loop, to get all the rows one after another. Let's look at an example where we do just that:

$query = mysql\_query("SELECT id, name, country FROM test\_users");

while($row = mysql\_fetch\_array($query))

echo $row["name"] . " is from " . $row["country"] . "<br />”;

We use a while loop, in which we assign the result of the mysql\_fetch\_array() function to the variable named $row on each iteration. This works because mysql\_fetch\_array() will return FALSE when there are no more rows, in which case the while loop will end. As long as there IS in fact a row left, mysql\_fetch\_array() places all columns and their values in the $row variable, where we can access it from simply by asking for the same name used in the SQL code. In our example, that gives us access to id, name and country, but to keep it less complicated, I have only used name and country so far.

So, that's the most basic ways of getting data out of the database. As you can see, it's fairly easy, since PHP has a bunch of nice functions for doing it, but there's much more to working with MySQL and there's definitely more to writing SQL.

## D. Testing queries

mysql\_connect("localhost", "username", "password");

mysql\_select\_db("my\_database");

$query = mysql\_query("Insert your SQL query here”);

while($row = mysql\_fetch\_array($query))

echo $row["name"] . " is from " . $row["country"] . "<br />”;

## E. Handling MySQL errors

When using PHP and MySQL together, you will likely run into a situation where you've made some sort of error in your SQL query, like misspelling a column name or a keyword or something like that. By default, PHP will not show you exactly what the problem is, only that you wrote a query which is not entirely correct. Let's try writing a faulty query to see the response from PHP:

$query = mysql\_query("SELECT id, namme FROM test\_users");

while($row = mysql\_fetch\_array($query))

echo $row['id'] . " - " . $row["name"] . " is from " . $row["country"] . "<br />”;

This is the example we are using a lot in this part of the tutorial, but in this case, we have misspelled the name column to provoke an error, which we get:

***Warning:*** *mysql\_fetch\_array(): supplied argument is not a valid MySQL result resource in test.php on line 7*

As you can see, the error is not thrown until we try using the resource returned by the mysql\_query() function, which we do when we call the mysql\_fetch\_array() function, in my file located on line 7. The error is very generic and not very helpful. This is on purpose, because knowledge about your database structure makes your website more vulnerable to SQL injection attacks, a problem we will discuss later on.

You might be able to spot the error and fix it in a lot of situations, but if not, you can use the mysql\_error() function to get a bit more information abut the problem. This function simply returns any error returned from the last executed MySQL function. You should only use this function for finding and fixing problems, and then remove it again once the problem has been fixed. Here's the above example, but where we call the mysql\_error() function to get more information:

$query = mysql\_query("SELECT id, namme FROM test\_users");

while($row = mysql\_fetch\_array($query))

echo $row['id'] . " - " . $row["name"] . " is from " . $row["country"] . "<br />";

echo mysql\_error();

This will give you a far more useful error message:

*Unknown column 'namme' in 'field list'*

## F. Cookies

A cookie is a little piece of information sent between the server and the client, which is most often a webbrowser. You can instruct the webbrowser to save your information and later retrieve it from the browser again. This allows you to save information about your visitors, for instance preferences or statistical information. A very common usage of cookies is to see whether or not a certain person has visited the site before, usually within a defined range of time, to tell whether the user should be counted as a new visitor or a returning visitor. Information like this is used for statistics, for instance a counter.

PHP can instruct the webbrowser to save a cookie by using the setcookie() method. Of course, the webbrowser can fail to do so for several reasons, for instance because the user has said no to cookies, but most modern webbrowsers will save the cookie for you. The setcookie() method allows you to define how long the cookie should be saved for and which path and domain it should be valid for.

However, in its most simple form, the setcookie() method is used like this:

setcookie("user\_name", "John Doe");

The first parameter is the name of the cookie, and the second is the value. In this case we store a cookie with the name "user\_name", with a value of "John Doe". The value can be retrieved again by using the $\_COOKIE superglobal, like this:

echo $\_COOKIE["user\_name"];

You should be aware that the setcookie() method is a header related function, which means that it has to be called before any output is made to the browser (including text, HTML tags and so on) and that the value you set can't be read until next time the page is loaded, which just means that you can't save a cookie and read the value in the same page execution. As mentioned, you can define for how long a cookie should be stored.

The third (and optional) parameter for setcookie() is used for this. If no parameter is supplied, or if it has been set to 0, the cookie will expire when the session ends, which typically is the same as when the browser is closed. Otherwise you should define the time when the cookie should expire, with a Unix timestamp. Here's an example where we set a cookie that will last for 1 hour:

setcookie("user\_name", "John Doe", time() + 3600);

We use the time() method to get the current time as a Unix timestamp and then we add an hour, which is the same as 3600 seconds.

Now let's look at a more complete example of how cookies can be used. First some code, then an explanation of it all:

**<?php**

if(isset($\_GET[“color"])) {

setcookie("color", $\_GET["color"]);

header("Location: " . $\_SERVER["PHP\_SELF"]);

}

if(isset($\_GET[“reset"])) {

setcookie("color", "");

header("Location: " . $\_SERVER["PHP\_SELF"]);

}

if(isset($\_COOKIE[“color"])) {

echo "Your favorite color is: " . $\_COOKIE["color"] . "<br />";

echo "<a href=\"?reset=1\">Click here to reset</a>";

}

else {

echo "What's your favorite color?<br /><br />";

echo "<a href=\"?color=red\">Red</a>&nbsp;&nbsp;";

echo "<a href=\"?color=green\">Green</a>&nbsp;&nbsp;";

echo "<a href=\"?color=blue\">Blue</a>&nbsp;&nbsp;";

}

**?>**

Okay, lots of lines here, but it's not too complicated though. With the first if statement, we check if the colour has been set in the query string. This happens when we click on one of the links that we output later. If a link has been clicked, it means that the user has made a favourite colour decision and we then call the setcookie() method to save it.

After that, we perform a little trick, to make sure that the user choice is reflected immediately: We do a redirect to the current, basically a reload of the page, using the header() method with the Location header. You don't have to completely understand this part, just know that we do it to be able to read the cookie immediately - otherwise the user would have to manually reload the page to see their choice reflected.

Since we also offer a reset option, we check for the reset parameter. If it has been set, the user wants to delete their previous choice and therefore we set a cookie with the same name, but with an empty value. This is the same as deleting the cookie. After that, we do the same redirect trick as described above.

After that, we finally get to the part where we check whether a cookie has been set or not. If it has, we remind the user which choice they made and then offer them a chance to reset it. If it hasn't been set, we ask the user which color is the favourite and then output some options to click on, which will activate the cookie setting code as first described.