

Hypertext PreProcessor (PHP)

PIC 40A, UCLA

©Michael Lindstrom, 2016-2022

This content is protected and may not be shared, uploaded, or distributed.

The author does not grant permission for these notes to be posted anywhere without prior consent.

PHP

PHP, an acronym for **PHP Hypertext Preprocessor**, is a popular programming language for the server-side of a website (roughly **80%** of websites use PHP on the server-side).

There was a time long ago when the acronym made more sense, standing for **Personal HomePage**.

For the most part, it is a series of scripts that generate HTML that a web browser displays, without the clients ever seeing the code that goes on behind the scenes.

Like JavaScript, it is a high level language. It is also an interpreted language: what you write as a programmer is parsed by a runtime engine.

PHP Topics I

We will look into the following PHP-related topics:

- ▶ PHP configuration
- ▶ PHP data types
- ▶ Math
- ▶ Arrays
- ▶ Coercions and type juggling
- ▶ Functions and closures
- ▶ Global variables
- ▶ Control flow
- ▶ Superglobal variables and webforms
- ▶ Cookie management
- ▶ File I/O
- ▶ PHP Sessions
- ▶ Sending emails

PHP Topics II

- ▶ File uploading
- ▶ JSON and communication between JS and PHP
- ▶ User-defined classes and Object Oriented Programming
- ▶ Copy on write optimization and passing by value vs reference
- ▶ Cross Origin Support
- ▶ HTTP Methods with PUT and DELETE

PHP Setup

PHP code is nested between **<?php** and **?>**, which is sometimes abbreviated as **<?** and **?>**. The first is more robust, however, so should be used.

For PHP to run on a webpage, the very first line should be a link to the binary that compiles the program! For many servers (and PIC) it is:

#!/usr/local/bin/php

If that is not there at the very top of the page, it will not run PHP!

PHP Setup

Many servers are based with a Linux operating system and the PHP engine may stumble and generate horrific errors if it runs across Windows-based return carriages, etc.

If a PHP file was written in a Windows editor, after being transferred to the server, it should be run through the Linux command:

dos2unix file_name.php

Or all files can be run like this together with:

dos2unix *

In addition, for PHP to run, they must have **755** (or above) permissions. When the server runs PHP scripts, it runs as **wwwrun** or some other "outside" user, and without proper permissions, the scripts can't run.

Simple PHP Program

Here is a simple PHP program to display the version of PHP run by the engine:

```
1  #!/usr/local/bin/php
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <title>PHP Demo</title>
6  </head>
7  <body>
8      <main>
9          <?php
10             # display our version of PHP
11             echo '<p>Current PHP version ', phpversion(), '</p>';
12         ?>
13     </main>
14 </body>
15 </html>
```

Current PHP version 5.5.14

Simple PHP Program

This PHP file looks a lot like a standard HTML file. At the top we link to the PHP binary. We write normal HTML code.

Interspersed in the code is PHP code, enclosed in the **<?php ... ?>** which produces more HTML that the browser has to parse.

Simple PHP Program

In PHP single line comments can be prefixed by either **#** (as in the example) or **//**.

Multiline comments begin with **/*** and end with ***/**.

The **phpversion** function outputs the current version of PHP being run.

Simple PHP Program

echo is not a function, but a language construct. It will print everything until there is a semicolon. It can take multiple inputs by separating them with a comma.

Closely related is **print**. It always returns the value 1. **print** can only take one argument.

Both **echo** and **print** can be called with or without parentheses.

Neither are real functions, which is why they can be called without parentheses.

Simple PHP Program

From a visitor's perspective, they really just see HTML. Here's what happens if we view the page source...

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>PHP Demo</title>
5  </head>
6  <body>
7    <main>
8      Current PHP version 5.5.14
9    </main>
10 </body>
11 </html>
```

Data Types

In PHP there are four scalar types:

- ▶ **boolean**
- ▶ **integer**
- ▶ **float**
- ▶ **string**

PHP also has

- ▶ **arrays**
- ▶ **objects**
- ▶ **NULL**
- ▶ **resources**

Variable Names

In PHP, **all variables begin with a \$**. They can then contain any number of underscores, letters, or digits.

```
$x = 42; // x is a variable storing 42
```

Warning: if the dollar sign is forgotten, all sorts of bugs can come up where things aren't printed, values aren't used, etc. All variables must have the dollar sign.

Data Types: Boolean, Integer, Float

In PHP, the boolean values are **true** and **false**. These reserved keywords are case insensitive: **True** and **False** also work, etc.

If a number is given without a decimal in it, it is taken to be an integer. With a decimal, it is a floating value.

```
$x = true; // boolean
```

```
$y = 11; // integer
```

```
$z = 0.29; // float
```

Data Types: Boolean, Integer, Float

All of the standard C++/JS operators exist in PHP: `+`, `-`, `*`, `/`, `%`, `++`, `--`, `+=`, `-=`, `*=`, `/=`, and `%=`.

For `+`, `-`, and `*` between two integers, the result is always an integer. For `/`, if the result would involve a decimal, it will be a float.

The mod operators `%` and `%=` coerce their arguments to integers if not already integers.

PHP Inputs

For the most part, PHP runs based on inputs provided to it through **get** and **post** methods. We look at a simple example here where a variable is set through a query string. Consider the file **index.php** at <http://www.website.com/>.

index.php

```
1  #!/usr/local/bin/php
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <title>GET Demo</title>
6  </head>
7  <body>
8      <main>
9        <?php
10         # retrieve value of x from query string
11         $x = $_GET['x'];
12         echo ' <p>Value of x is: ', x, ' </p>';
13       ?>
14     </main>
15 </body>
16 </html>
```

PHP Inputs

By a user visiting the landing page with a query string, they can change what the page displays. For example,

`http://www.website.com/index.php?x=7`

will produce a page with a paragraph saying “Value of x is: 7.”

A query string populates a special variable **`$_GET`**, which we discuss later. We can retrieve values that were set through the query string by subscripting **`$_GET`**.

PHP Inputs

Remark: simply by appending a query string **?name=value**, etc., to the end of a PHP script name, the **\$_GET** superglobal is populated with the property names and values.

www.someplace.com/somepage.php?name=foo

```
1  #!/usr/local/bin/php
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <!-- page title will be foo -->
6      <title><?php echo $_GET['name']; ?></title>
7  </head>
8  </html>
```

Math

PHP has some functions that are defined in a global namespace, including:

- ▶ **abs** (absolute value)
- ▶ **sin, cos, tan** (sine, cosine, tangent)
- ▶ **exp, log** (natural exponential, natural logarithm)
- ▶ **pow** (raises first argument to power of second argument)
- ▶ **round** (rounds to nearest integer)
- ▶ **ceil, floor** (integer ceiling and floor functions)

Math

Random number in PHP require multiple stages. First, we must **seed** the random function:

```
mt_srand();
```

The random numbers will be integers that are uniformly distributed from 0 to some maximum value. That maximum value is returned from **mt_getrandmax**:

```
$max_rand = mt_getrandmax();
```

To get a random integer in the range, we use **mt_rand**:

```
$val = mt_rand() / ($max_rand+1); // will be from 0 to 1
```

```
$die_roll = floor(6*$val) + 1;
```

Data Types: String

Strings are usually (some bizarre exceptions exist) enclosed in either single or double quotes. The two are not always the same. For plain text, the single quote is preferred because content within double quotes goes through a more thorough parsing.

The `.` operator concatenates strings.

```
$x = 'hello, ';  
$y = 'world!';
```

```
echo $x, $y; // prints "hello, world!"
```

Data Types: String

Escape sequences are permitted within strings.

For a single quoted string, the only permitted escape sequences are `\'` and `\\` for a `'` and `\`, respectively. All other backslashes are interpreted literally.

Double quotes allow for more escape sequences like `\'` for `"`, `\t` for a tab, `\n` for a new line. But they also allow variables to be evaluated within the string.

```
$year = 2022;  
echo "The year is $year."; // prints "The year is 2022."  
echo 'The year is $year.'; // prints "The year is $year."
```

To print `$` within double quotes, we use `\$`.

Data Types: String

Strings are indexed from 0.

```
$msg = 'hello world';  
echo $msg[4]; // prints "o"
```

There are a number of handy functions to use for strings:

```
echo substr($msg, 6); // prints "world";
```

The **substr** takes a string argument, a starting index, and an optional length. Its behaviour is similar to that of C++ and JavaScript, but we are not calling a member function.

Data Types: String

```
$msg = 'hello world';
```

```
strlen($msg); // 11
```

```
strpos($msg, 'world'); // 6
```

strlen gives the length of a string.

strpos searches a string from the beginning for a piece of text returning the index where it begins. If the string is not found, it returns **FALSE**.

Remark: obviously this can be dangerous. Indeed **FALSE** can very easily be converted to **0**. As with JavaScript, we will need **===** and **!==** operators.

Data Types: String

String concatenation is done via the `.` and `.=` operators with the latter including assignment.

```
$x = 'ab';  
$y = 'c';  
$z = $x . $y; // $z === 'abc'
```

```
$msg = 'hello';  
$msg .= ' world'; // now $msg === 'hello world'
```

Remark: yes, there are `===` and related comparisons in PHP...

Data Types: String

```
$msg = 'hello world';
```

```
$arr = explode(' ', $msg); // array, ['hello', 'world'];  
$msg2 = implode(' ', $arr); // string 'hello world'
```

The **explode** function takes a delimiter and a string, splitting that string into parts of an array.

The **implode** function takes a delimiter and an array, making a string out of it, with that delimiter placed between elements.

Data Types: String

Using the subscript for strings appears to allow a single character to be changed.

```
$n = '123';  
$n[1] = '0'; // now n===103  
$n[1]='456'; // now n===143
```

Data Types: Constants

PHP allows for scalar types to be kept constant. We use the **define** syntax:

```
define('FOO',19);
```

Above, we defined the symbol **FOO** to be the value 19.

Constants are often capitalized by convention. Valid names begin with a letter or underscore, and then contain any number of letters, digits, or underscores.

Data Types: Constants

In newer versions of PHP, it is also possible to use the **const** keyword to define a scalar type and write

```
const FOO = 19;
```

In C++, this is just like specifying a preprocessor directive with define:

```
#define FOO 19
```

Aside: in the newer versions of PHP, **const** can even be used to define an array.

PHP Comparisons

Logical negation is done with `!`.

We use `<` for less than, `<=` for less than or equal, `>` for greater than, `>=` for greater than or equal.

With `==`, two values are **equal**, after possible coercion, and both `!=` and `<>` mean two values are not equal, even after coercion.

In the case of arrays, `==` checks if they are of the same length and their respective values are equal (up to coercions).

PHP Comparisons

With `===`, two values are **identical**, i.e., equal and of the same type and `!==` signifies either the types differ or the two values are not equal. For objects, `===` evaluates to **true** if they reference the same object in memory.

In the case of arrays, `===` checks if they are of the same length and their respective values are equal and of the same type.

This **identical** can go a bit too far. In PHP, `0 === 0.0` is **false** because we are comparing an integer and a floating point!

PHP Coercions

PHP uses the same sorts of coercions as JS for the most part in determining whether something is **truthy** or **falsey** and when comparing between different types it can try to convert them to numbers. But with PHP, "0" is **falsey** and an empty array is also considered **falsey**.

Some other fascinatingly annoying coercions: the booleans **true** and **false** can be coerced into the strings '1' and '' (empty string!). Thus if we ever **print** or **echo false**, nothing gets printed...

In PHP these coercions are often called **type juggling**.

PHP Coercions

When trying to convert a string to a number, PHP will extract the numeric part of the string starting from the beginning, ignoring initial white space, stopping when it reaches letters that don't make sense. But it will read scientific notation, too. And the empty string is converted to 0.

In PHP, **"php" == 0** is **true!!!**

PHP processes **+**, **-**, *****, **/**, **%**, and **.** from left to right. With **.** it will convert the operands to strings; otherwise it will convert the operands to numbers.

PHP Coercions

`3 + '5hello' . 7; // '87': (3+5) . 7 => 8 . 7 => '87'`

`12 . 3 - '4'; // 119: (12 . 3) - '4' => '123' - '4' => 123-4 = 119`

`'hello6' + 7; // 7: 0 + 7`

`'4hello' * 7; // 28: 4 * 7`

`'1.2e+2hello' + 7; // 127: 120 + 7`

`'hello' . 7; // 'hello7': concatenation`

`4 . 7; // '47': concatenation`

PHP Coercions

To view the type of a variable, we use **gettype**:

```
$x = true;
```

```
$y = 'red';
```

```
gettype($x); // boolean
```

```
gettype($y); // string
```

Data Types: Array

Like with JavaScript, PHP arrays can store anything, even functions. We can use the **array** function to create an array (or use square brackets as shown later):

```
$x = array(4,true,'hello'); // stores value 4, true, and 'hello'
```

And, like strings, arrays have 0-based indexing.

```
echo $x[2]: // prints "hello"
```

Data Types: Array

Some useful functions with arrays:

```
$x = array('c','b','a');  
$new_length = array_push($x, 'z', 'y'); // $x = ['c','b','a','z','y'];  
$last_entry = array_pop($x); // $x = ['c','b','a','z']  
echo count($x); // 4  
sort($x); // sorts the array alphabetically (all items are strings)
```

The **array_push** function accepts an array and an arbitrary list of values to add to the end, returning the new length.

The **array_pop** accepts an array and removes its last element, returning that value.

The **count** function accepts an array and returns its number of elements.

The **sort** function accepts an array and sorts it in ascending order.

Data Types: Array

Remark: if **sort** is called on an array with a mix of data types, the sorting is apparently not stable.

```
$x = array(2., 1, false, '0', '-19');  
sort($x); // ['0', false, '-19', 1, 2.]
```

To do more complicated sorting, we need to use the **usort** function...

Data Types: Array

Secretly all PHP arrays are **associative**, i.e., having key-value pairs. By default the keys are just indices. The keys can be either **strings** or **integers**.

```
$arr = array('foo'=>'bar', 'cat'=>'meow', 4=>7, 9.8=>false);
```

```
// so we have
```

```
$arr['foo']; // 'bar'
```

```
$arr['cat']; // 'meow'
```

```
$arr[4]; // 7
```

```
$arr[9]; // false;
```

We can also add new elements to this array by subscripting it:

```
$arr['colour'] = 'red';
```

Data Types: Array

A newer piece of syntax, more consistent with JavaScript is creating an array with square brackets like:

```
$x = [2,4,6,true];
```


Data Types: Array

There is also the `[]` syntax to add an element at the end of an array. It is more efficient than calling **array_push** if only a single element is being added.

```
$x = [1,2,3];  
$x[] = 4; // now last element is 4
```

To remove an element from an array, we use **unset**:

```
unset($x[2]); // now $x = [1,2,4], removed item index 2
```

Warning: this can have weird effects upon the array indices.

Remark: a variable can be deleted and removed from the namespace by calling **unset** upon it.

Data Types: Array

For arrays indexed by integers, the simple remedy to the “missing” indices when an element is **unset** is to use the **array_values** function.

```
$x = ['a', 'b', 'c'];
```

```
unset($x[1]); // so now $x is { 0=>'a', 2=>'c'}
```

```
$x = array_values($x); // now $x is { 0=>'a', 1=>'c'}
```

Data Types: Object

An **object** is the equivalent of the C++ class structure. Objects can store member variables and member functions.

Given an object handle, properties and methods are accessed via -> (the equivalent of the . access in JavaScript and C++).

```
$x = new Square(3);  
echo $x->get_area();
```

We will study how to write them later.

Data Types: NULL

NULL is the special value and the data type **null** that represents the state of not having a value. This is like the JavaScript **null** or **undefined**.

```
$a = NULL; // has no value
```

Data Types: Resource

A **resource** is a variable that references an external resource (file).

File pointer resources (like the C++ **std::ifstream**, **std::ofstream**) allow us to read from and write to files would be one example. A link to a **database** through which we can make queries would be another example.

Type Juggling

In PHP, we can explicitly cast one variable type to another using the C-style cast syntax. The following conversions are supported by PHP: conversion to **bool**, boolean; **int**, integer; **float**, float; **string**, string; and **array**, array.

For example:

```
$x = 4.14;
```

```
$y = false;
```

```
$a = (int) $x; // $a is 4
```

```
$b = (string) $y; // $b is ""
```

```
$c = (bool) 'hello'; // $c is true
```

var_dump

For debugging purposes, it can be handy to display the value of a variable. For that, we can use **var_dump**.

```
$x = "hello";  
$y = 13;  
$z = array(4,true);
```

```
var_dump($x);  
var_dump($y);  
var_dump($z);
```

```
string(5) "hello"
```

```
int(13)
```

```
array(2) { [0]=> int(4) [1]=> bool(true) }
```

Functions

The basic syntax to create a function in PHP is the same as in JS:

```
function name_of_function( list_of_parameters ) {  
    what_to_do;  
    return something; // or do not return if not returning a value  
}
```


References vs Values

PHP supports both passing by reference and passing by value.

In code, to make one variable a reference to another, we use the **&** operator:

```
$x = 11;  
$y = &$x;  
++$y; // both $x and $y are 12
```

References vs Values

For a function, the input parameters need to have an & to accept by reference.

```
function does_nothing($x) { ++$x; }
```

```
function does_something(&$x) { ++$x; }
```

Function Default Arguments

PHP supports scalar values, arrays, and **NULL** values as default arguments for functions.

```
function foo($x, $y = array(1,2,3), $z = 111){  
    echo $x + $z;  
}
```

Note: default arguments must be specified from right to left.

```
foo(3); // 114  
foo(3,4); // 114  
foo(3,4,5); // 8
```

```
foo(); // PHP Warning: missing 1 argument for foo...
```

Global Variables in Functions

In PHP, if a variable is defined in a function, it is only known within that scope. Also, only variables passed as parameters are known to the function by default. It will not be able to access a global variable unless we request so.

```
$some_global = 111;
```

```
function foo($y) {  
    global $some_global; // requesting access to the global  
    $z = $some_global + $y; // this is okay  
    echo $z;  
}
```

```
echo $z; // ERROR: z not defined here!
```

Within a function body, a new global variable can also be created with *global \$name_of_variable = something;*

Global Variables in Functions

Alternatively, we can access the value from the **\$GLOBALS** variable:

```
$some_global = 111;  
function bar($y) {  
    echo $GLOBALS['some_global'] + $y;  
    $GLOBALS['z'] = $GLOBALS['some_global'] + $y; // adds to globals  
}
```

The **\$GLOBALS** variable stores all global variables. Above, we could access **\$some_global** by the index '**some_global**' without the **\$**. We also added **\$z** as a global variable so that the code below is okay:

```
echo $z;
```

Anonymous Functions

In PHP, there are also function objects that can be created on the fly. Here are a few examples too see the syntax.

```
1  $fun = function($array, $name){ // anonymous function stored as $fun
2      $array[0]();
3      $array[1]($name);
4  };
5
6  $arr = array( // this array stores functions
7      function() { echo 'hello '; },
8      function($name) { echo $name; }
9  );
10
11 $fun($arr, "Joe"); // prints "hello Joe"
```

Anonymous Functions

Just as in JavaScript and C++, we can use anonymous functions as call arguments to sort lists.

```
1  $names = array('Colleen', 'Bryce', 'Adam', 'Dan', 'Ella');
2
3  // we will sort by name length!
4  usort($names,
5       function($x,$y)
6         { return strlen($x) - strlen($y); }
7       );
```

The **usort** function takes an array to sort as its first argument and a function as its second argument. The comparison function *must* return a negative integer, zero, or positive integer, respectively, depending on whether its first argument is less than, "equal to", or greater than the second argument, respectively, based on our sorting criteria.

Remark: the **integer** fact is important. Floating points may not work as they will be cast to integers.

Anonymous Functions

Alternatively, we could have written a free function and passed it as an argument in quotes.

```
1  function sort_lengths($x,$y){
2      return strlen($x) - strlen($y);
3  }
4
5  $names = array('Colleen', 'Bryce', 'Adam', 'Dan', 'Ella');
6
7  // we will sort by name length!
8  usort($names, 'sort_lengths');
```

Function Documentation

We will focus on a simple documentation style similar to C++ and JS.

```
/**  
description  
  
@param type $name description  
...  
@return description  
*/
```

```
1  /**  
2  This function compares the lengths of two strings  
3  
4  @param string $x the first string  
5  @param string $y the second string  
6  
7  @return the length of $x minus the length of $y  
8  */  
9  function sort_lengths($x,$y){  
10     return strlen($x) - strlen($y);  
11 }
```

Functions Accepting or Returning References

PHP adheres to a **copy on write** optimization. This means that passing by reference is only required if the intent is to modify the argument! It is not done for efficiency.

PHP will pass arguments by reference and only if they are modified within the function does that turn into a pass by value. **But objects acquired through “new” behave like pointers so there are subtleties.** More to come in the object oriented discussion.

In C++, we very often pass an object to a function as a reference to const, even if it is not being modified to avoid an unnecessary copy of an object.

```
bool checks_string(const std::string& input);
```

Function Arguments

Like JavaScript, PHP can accept arbitrary numbers of arguments. Within a function body, **func_num_args()** returns the number of arguments that were passed. And **func_get_args()** returns an array of the values passed.

```
function foo() {  
    $len = func_num_args();  
    $inputs = func_get_args();  
  
    for($i=0; $i<$len; ++$i){  
        var_dump($inputs[$i]);  
    }  
}
```

Function Ordering

Like JavaScript, a function need not be declared/defined before it is referenced and used. But it should be defined somewhere.

// works fine:

```
function foo() { bar(); }
```

```
foo();
```

```
function bar() { echo '11'; }
```

Closures

PHP allows for **closures** but we need to request the closure by specifying the capture parameters with **use (list of parameters)**.

```
function transform($m,$b) {  
    return function($x) use ($m,$b) { return $m * $x + $b; }  
}
```

```
$celsius_to_fahrenheit = transform(1.8, 32);  
$winnipeg_mild_winter_in_F = $celsius_to_fahrenheit(-25); // -13
```

With C++ the "use" appears as the capture list in square brackets:

```
auto transform(double m, double b) {  
    return [m,b](double x)->double{ return m*x+b; };  
}
```

Control Flow: if/else I

All programming languages need to have control flow and here we'll look at the control flow structure of PHP. We start with **if** statements:

```
if(condition) {  
    // do stuff  
}
```

```
if(condition) {  
    // do this  
}  
else{  
    // or this  
}
```

Control Flow: if/else II

```
if(condition) {  
    // do this  
}  
elseif{  
    // this  
}  
else {  
    // or this  
}
```

In PHP, **elseif** is a thing, meaning the same thing as **else if** but it applies more widely than **else if**.

Control Flow: if/else

Warning: using correct variable names is really important in PHP. Forgetting a dollar sign can be an absolute nightmare!

```
$reveal_deep_dark_secret = false; // don't share it!!!
```

```
if( reveal_deep_dark_secret ) {  
    echo 'revealing secret now...';  
}
```

Above, the secret gets revealed. The missing \$ in front of **\$reveal_deep_dark_secret** in the **if** condition ends up treating the condition as the string 'reveal_deep_dark_secret' which is coerced to **true** because it is nonempty and not '0'...

Control Flow: if/else

More warnings: Here's another thing to be wary of... let's suppose you did remember the \$ but made a typo somewhere else...

```
$did_good_job = true;

if($did_goodjob){ // oops, typo..., definitely not true now...
    echo 'well done!';
} else{
    echo 'hang your head in shame!';
}
```

Something PHP knows is a variable but which has not been set yet is treated as **NULL** (which is coerced to **false**).

Control Flow: for/while/do

There are really no surprises for these loops.

```
for(initializations; condition; after body steps) {  
    // do stuff  
}
```

```
while(condition) {  
    // do stuff  
}
```

```
do {  
    // stuff  
} while(condition);
```

Control Flow: foreach

Like the C++ range-for and JS **for of**, PHP has **foreach** to iterate through arrays and objects.

```
$arr = ['a'=>1, 'b'=>2, 'c'=>3];
```

```
// with this syntax, we go through the values 1, 2, 3 calling them $val  
foreach($arr as $val) { // prints: "1 2 3"  
    echo $val, ' ';  
}
```

```
// with this syntax, we track the keys and values  
// for each iteration, the key is $key and the value is $val  
foreach($arr as $key=>$val) { // prints: "a=>1 b=>2 c=>3"  
    echo "$key=>$val ";  
}
```

Note: we used double quotes above to evaluate the variables.

Control Flow: foreach

Remark: there is nothing special about the names **\$key** and **\$val**: they could be named anything. They are names we use in the loop.

Unfortunately in PHP, variables defined within a loop are also defined after a loop executes...

Control Flow: foreach

Using an **&** allows us to go through the elements by reference but we must remember to **unset** the loop variable. Generally variables defined in loops will live beyond the loop...

```
// with this syntax, we change the array's values
```

```
foreach($arr as &$amp;val) { // prints: "1 2 3"
```

```
    $val *= 2;
```

```
}
```

```
// WARNING: $val currently still references the last element of the array!
```

```
unset($val); // break $val from referencing last element
```

Control Flow: foreach

Loops with key value pairs also admit the values to be references, e.g.,

```
foreach($some_array as $k=>&$v) {  
    ++$v;  
}
```

```
unset($v);
```

Control Flow: foreach

As in C++ and JS, **if**-statements, **for**, **while**, and **foreach** loops do not require braces to be grammatically correct if the body is a single statement. But for robustness, they should be included! Also each nontrivial control flow branch should be commented.

Control Flow with Colons

PHP has an alternative syntax for control flow. Instead of, for example:

```
if($x) {  
    ++$y;  
    echo $y;  
} else {  
    --$y;  
}
```

we could write:

```
if($x) :  
    ++$y;  
    echo $y;  
else:  
    --$y;  
endif;
```


Control Flow with Colons

With this, braces are not required, even for multiple statements in a control flow branch. Each opening brace is replaced by `:`.

But without braces, it's impossible to know when a block ends, so the parser searches for a follow-up control flow keyword like **else**... and at the end of the day, something has to signify the end of the control flow structure so there is, as in this example, and **endif**.

The requisite endings are **endif**;, **endwhile**;, and **endforeach**;

And when using the colon syntax, **else if** is not allowed but **elseif** is.

Control Flow with Colons

Danger: while **elseif** and **else if** work fine for the normal braced control flow structures, **else if** is not valid for control flow structures that use the colon.

Danger: do not mixed the syntaxes of the colon or non-colon control flow. It is an error to mix the two syntaxes in a single control flow statement; it is also likely to throw an error to mix the syntaxes between inner and outer branches of control flow structures.

Control Flow: Interspersing HTML and PHP

We can intersperse HTML and PHP. Below, **"ZYX"** gets printed. Then 10 lines of **"hello"**.

```
1  #!/usr/local/bin/php
2  <?php
3  $str = "0";
4  if($str){ // recall that the string "0" is falsey in PHP ?>
5      <p>ABC</p>
6      <?php
7  } else { // so execute this ?>
8      <p>ZYX</p>
9      <?php
10 }
11
12 for($i=0; $i<10; ++$i) { // do this 10 times ?>
13     <p>Hello</p> <?php
14 }
15 ?>
```

Effectively HTML without being contained inside of PHP tags is treated like an **echo** and just gets rendered directly.

HTTP Requests

HyperText Transfer Protocol (HTTP) requests allow for clients to request information/data from a server.

With a **get** method, a query string is generated and appears as part of the URL the user sees. It is quite common, but is not secure and sensitive data like passwords, etc., should never be used with it. A **get** method can only be used to request data, never to modify it.

With a **post** method, a query is sent, but it is "packed in a sealed envelope", as a kind of analogy. No one can see the request. These methods can create/modify data on the server.

There are also **put**, **delete** methods.

Superglobals - \$_SERVER

We have already encountered **\$GLOBALS**, one of the superglobals of PHP. In managing HTTP requests, a few more become important.

\$_SERVER stores information about file names, script locations, etc. Two of its most useful members are:

\$_SERVER['PHP_SELF'], the current php page/script, and

\$_SERVER['REQUEST_METHOD'], the method of access: GET, POST, etc. For example,

```
if ($_SERVER['REQUEST_METHOD'] === 'POST') { /* do stuff */ }
```

Superglobals - \$_POST

\$_POST stores data from form when **method=post** is used. We access form data by indexing this variable for the names of the form elements!

```
1  <form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>" >
2      <input type="text" name="userMessagePost" />
3      <input type="submit" value = "post" />
4  </form>
5
6  <p>
7  <?php
8      echo 'Post: ', $_POST['userMessagePost'], '<br/>';
9  ?>
10 </p>
```

Post:

Post: foo

Superglobals - \$_POST

A remark is worth remarking on the **action=** part of the web form.

In a web form, an attribute **action** can be specified. This action can be a few lines of JavaScript or a link to a script to run, PHP or JavaScript.

If we wanted to run a PHP script on another page, we could write **action="that_other_page.php"** and redirect the user and their information over there.

To allow the PHP script to run on the current page, we use **\$_SERVER['PHP_SELF']** to reference the current PHP script. And by **echoing** that in the php script in the quotes, we refer the action to the same page.

Superglobals - \$_GET

\$_GET stores data from form when **method=get** is used. We access form data by indexing this variable for the names of the form elements!

With **get**, we see the name-value pairs appearing in the URL in the form:

somephpscript.php?var1=value1&var2=value2

```
1  <form method="get" action="<?php echo $_SERVER['PHP_SELF']; ?>" >
2    <input type="text" name="userMessageGet" />
3    <br/>
4    <label for="A">A</label> <input type="radio" name="letters" value="A" id="A" />
5    <label for="B">B</label> <input type="radio" name="letters" value="B" id="B" />
6    <br/>
7    <input type="submit" value = "get" />
8  </form>
9
10 <p>
11 <?php
12     echo "Get: ";
13     if(isset($_GET['userMessageGet'])){ // check if set
14         echo $_GET['userMessageGet'], '<br/>';
15     }
16     if(isset($_GET['letters'])){
17         $_GET['letters'], '<br/>';
18     }
19     ?>
20 </p>
```

Superglobals - \$_GET

hello world

A ☐ B ☒

`http://www.page.com/folder/index.php`

Get:

A ☐ B ☐

Get: hello world

B

**`http://www.page.com/folder/index.php?
userMessageGet=hello+world&letters=B`**

Arrays in Forms I

Recall that for web forms that have checkboxes or select fields where multiple values can be submitted, it is necessary to give a **name="something[]"** for that collection of options. Using the array syntax ensures that an array is captured. The array could be empty if no options are selected. If multiple options are selected, there will be an array carrying all the values that were selected.

Arrays in Forms II

```
1  <form id="byGet" method="get" action="<?php echo $_SERVER['PHP_SELF']; ?>">
2    <input type="checkbox" name="x[]" value="A">A<br/>
3    <input type="checkbox" name="x[]" value="B">B<br/>
4    <input type="checkbox" name="x[]" value="C">C<br/>
5    <input type="submit" value="GET" />
6  </form>
7
8  <p>
9  <?php
10     if(isset($_GET['x'])){ // ensure was set
11         $checks = $_GET['x']; // store array as $checks
12         if (count($checks)==0){ // size 0 means no checks
13             ?>
14             <h2>Nothing was checked</h2>
15         <?php
16         }
17         else{ // something was selected
18             foreach($checks as $selected){ // go through each selection
19                 echo $selected, " was chosen. ";
20             }
21         }
22     }
23     ?>
24 </p>
```

Arrays in Forms

<input type="checkbox"/> A <input checked="" type="checkbox"/> B <input checked="" type="checkbox"/> C <input type="button" value="GET"/>
--

Nothing was checked

<input type="checkbox"/> A <input type="checkbox"/> B <input type="checkbox"/> C <input type="button" value="GET"/>
--

B was chosen. C was chosen.

Superglobals - \$_COOKIE I

In PHP, there is a global object **\$_COOKIE** that has access to cookie data. Here is a simple example.

page.php

```
1  #!/usr/local/bin/php
2  <?php
3      if( isset($_POST['add_php']) ){ // PHP should set cookie
4          setcookie('type', '', time()-1 ); // remove what is there
5          setcookie('type', 'php', time()+60);
6          header('Location: cookie.php'); // send user back to page to see effect
7          exit;
8      }
9      if( isset($_POST['remove_php']) ){ // PHP should forget cookie
10         setcookie('type', '', time()-1 );
11         header('Location: cookie.php'); // send user back to page to see effect
12         exit;
13     }
14     ?>
15     <!DOCTYPE html>
16     <html>
17     <head>
18         <title>Cookies in PHP and JS</title>
19         <script src="cookie.js" defer></script>
20     </head>
21     <body>
22         <p>PHP Cookie: <?php
23         if( isset($_COOKIE['type']) ){
24             echo $_COOKIE['type']; // state the value of 'type' of within cookie
```

Superglobals - \$_COOKIE II

```
25     } ?></p>
26 <p id="show_js"></p>
27 <form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>" >
28     <input type="button" id="add_js" value="Add with JS"/>
29     <input type="button" id="remove_js" value="Remove with JS"/>
30 </br>
31     <input type="submit" name="add_php" value="Add with PHP"/>
32     <input type="submit" name="remove_php" value="Remove with PHP"/>
33 </form>
34 </body>
35 <html>
```

cookie.js

```
1  document.getElementById('add_js').addEventListener("click",
2      function(){ // for JS to set the cookie
3          const val="type=js;"
4          const now = new Date();
5          now.setMinutes( now.getMinutes() + 1 );
6          const expires = "expires=" + now.toUTCString() + ";";
7          document.cookie = val+expires; // no path given means current path
8          window.location = "cookie.php"; // send user back to page to see effect
9      }
10 );
11
12 document.getElementById('remove_js').addEventListener("click",
13     function(){ // JS removes the cookie
14         const val="type=";
15         const past = new Date(1);
16         const expires = "expires=" + past.toUTCString() + ";";
17         document.cookie = val+expires; // no path given means current path
```

Superglobals - \$_COOKIE III

```
18     window.location = "cookie.php"; // send user back to page to see effect
19 }
20 );
21
22 window.addEventListener("load",
23     function(){ // try to find the value of 'type' within the cookie
24         const cookie = document.cookie.split(';');
25         let val = '';
26         for(const part of cookie){
27             const arr = part.split('=');
28             // nonzero length and match with whitespace removed from ends
29             if( (arr.length > 0) && (arr[0].trim() === 'type' ) ){
30                 val = arr[1];
31             }
32         }
33         document.getElementById('show_js').innerHTML = "JS Cookie: " + val;
34     }
35 );
```

Superglobals - \$_COOKIE

PHP Cookie:

JS Cookie:

Add with JS	Remove with JS
Add with PHP	Remove with PHP

PHP Cookie: js

JS Cookie: js

Add with JS	Remove with JS
Add with PHP	Remove with PHP

PHP Cookie: php

JS Cookie: php

Add with JS	Remove with JS
Add with PHP	Remove with PHP

Top: when page loads and/or when a Remove button has been pressed.
Middle: when the JS Add button has been pressed.
Bottom: when the PHP Add button has been pressed.

Superglobals - \$_COOKIE

There is a lot going on in this example. It illustrates many subtleties to working with cookies and the differences between PHP and JS handling of cookies.

For PHP, any calls to **setcookie**, a function that sets cookies, must be handled "before any headers are sent". Headers are sent when HTML content, including whitespace, is rendered. The headers are part of the HTTP protocol specifying important information about what information the browser is receiving, etc. Cookie setting *must* come even before **<!DOCTYPE html>!** Failing to do so will cause the page to crash.

Superglobals - \$_COOKIE

In PHP the **time** function returns the number of seconds since January 1, 1970, 00:00 GMT time. The **setcookie** accepts arguments of name, value, an expiry time counting seconds since **time 0**, and a location where the cookie is valid. If no location is given, the default location is the current folder. If no time is given, the cookie expires when the browser closes.

Through headers, we can redirect the user. Setting the header also “sends headers.” With **header('Location: url')**, the page redirects to **url**. We do this so we can see the effects “immediately.” The PHP **\$_COOKIE** variable is populated only when the page first loads and if we did not redirect, we would need to refresh the page in order of PHP to observe the effects of the cookie.

Additionally, the buttons for the JS-end are not submit buttons so we redirect with the JS, too.

Superglobals - \$_COOKIE

The JS string function **trim** returns a string without whitespace on either end.

If we had wanted to see the effect of the cookie changes immediately on the PHP end when setting/removing with PHP, we would need to also assign to or **unset** the **\$_COOKIE['type']** value. But JS does not respond to that.

Reading and Writing Files

The **fopen** function gives us a file pointer resource, allowing us to open a file, either to read from it or to write to it. The flag "**r**" indicates reading, "**w**" indicates writing from the beginning and emptying the file, and "**a**" indicates appending to the end.

If the file does not yet exist, both "**w**" and "**a**" will attempt to create the file.

```
$file = fopen('file_name.txt', 'w'); // example of opening to overwrite.
```

Reading and Writing Files

If **fopen** fails, it returns the value **false** rather than a file pointer resource. Thus, it is common to see code such as:

```
$file = fopen('foo.txt', 'r') or die('could not open file');
```

The **or** keyword pertains to logical or, ||. If the opening fails then the first expression is false and the second one is evaluated. The **die** function causes the script to **exit** (a termination function) and displays an error message.

Reading and Writing Files

Remark 1: a resource evaluates to **true** and lazy logical evaluation is applied.

Remark 2: in PHP, **or** has lower precedence than **=** (which is lower than **||**). So the expression should be seen as:

- ▶ Evaluate **fopen('foo.txt', 'r')** and assign it to **\$file**.
- ▶ If that result is a file handling object, which is **truthy**, the lazy evaluation of **or** makes the **or** evaluate to **true** without further processing.
- ▶ And if **\$file** is **false** then the second term of the **or** is evaluated and the program ends.

Reading and Writing Files

The **fwrite** function takes a file pointer resource and content as arguments and writes them to the file.

```
$file = fopen('poem.txt', 'w') or die('cannot open');  
fwrite($file, "It is morning now.\nWeb programming is good fun!\nThis  
was a haiku");
```

We generate the file **poem.txt**:

```
It is morning now.  
Web programming is good fun!  
This was a haiku.
```

Remark: the double quotes are important here! With single quotes, line characters would be directly rendered as `\n`.

Reading and Writing Files

After using a file pointer resource, it should be closed to avoid having unused resources hanging about. We use **fclose** for that.

```
fclose($file);
```


Reading and Writing Files

The function **feof** returns a boolean value to indicate if the end of file has been reached.

The **fgets** function reads and extracts one line of a file at a time (up to and including a new line character). We can manage this white space with **trim**: it takes a string input and returns a string without whitespace at the ends.

```
$display_poem = fopen('poem.txt', 'r') or die('cannot open file');
```

```
while(!feof($display_poem)) { // while still more to read
    $line = fgets($display_poem);
    echo $line, '<br/>';
}
```

Flat File Management

A **flat file** as opposed to something like a **database** is one that stores data in plain, unformatted text, with nothing but forms of whitespace to designate separate pieces of data.

Working with data this way is often a lot simpler but it does come with an overhead of deciding on a format.

As a simple guideline, each record should occupy a single "line" – something that terminates in a new line character. And between field values, tabs can separate data.

The above is very clear: each entire line is one record. And a tab always means we are moving onto another field.

Flat File Management

When we read data from a flat file, we can easily read one record at a time with the **fgets**. But if the fields are separated by tabs, we should also remember the **explode** function to break the string apart.

```
$line = fgets($file);
```

```
$fields = explode("\t", $line);
```

Superglobals - \$_SESSION

Instead of using **cookies** on the user's end, we can store information on the **server** end with **PHP sessions**.

This entails preceding all HTML with a **session_start();** command. Like cookie setting, session setting must occur before headers are sent.

We can add values to the global **\$_SESSION** variable. Other PHP scripts that are accessed from a session page also get access to the session variables if they have a **session_start();** command at the top.

The **session_unset();** command clears session variables and **session_destroy();** command ends a session.

Superglobals - \$_SESSION

PHP sessions can (and often should) be named to avoid clashes with other PHP sessions. To name (or resume) a PHP session, we use:
session_name('whatever');

The **session_name** function must precede **session_start**. Then the session being started is either given the name prescribed or the session with the given name is resumed.

Simple Login System

Here we'll consider a very simple login system built with PHP. We assume there is a flat file **password.txt** on the server storing the password. Later with hashing and databases this can be made more secure.

When the user first visits **login.php**:

If they enter the wrong password:

Invalid password!

If their password is correct they are taken to **welcome.php**:

Welcome!

Simple Login System I

login.php

```
1  #!/usr/local/bin/php
2  <?php
3  session_save_path(dirname(realpath(__FILE__)) . '/sessions/');
4  session_name('Demo'); // name the session
5  session_start(); // start a session
6  $_SESSION['loggedin'] = false; // have not logged in
7
8  /**
9   This function validates a password and sets the $_SESSION token to true
10  if it is correct, logging them in and sending them to the welcome page.
11  Otherwise it flags $error as true.
12
13  @param string $password the password the user entered
14  @param boolean $error the error flag to possibly change
15  */
16  function validate($password, &$amp;error){
17      $fin = fopen('password.txt', 'r'); // open file to read
18      $true_pass = fgets($fin); // get the line
19      fclose($fin); // close the file
20      $true_pass = trim($true_pass); // trim white space
21
22      if($password === $true_pass){ // if they match, great
23          $_SESSION['loggedin'] = true;
24          header('Location: welcome.php');
25      }
26      else { // bad password
27          $error = true;
28      }
29  }
30
```

Simple Login System II

```
31     $error = false;
32     if(isset($_POST['pass'])){ // if something was posted
33         validate($_POST['pass'], $error); // check it
34     }
35     ?>
36 <!DOCTYPE html>
37 <html lang="en">
38 <head>
39     <title>Login Page</title>
40 </head>
41 <body>
42     <main>
43         <form method = "post" action = "<?php echo $_SERVER['PHP_SELF']; ?>">
44             <input type="password" name="pass" /> <input type="submit" value="log in" />
45             <?php if($error) { // wrong password ?>
46                 <p>Invalid password!</p> <?php
47             } ?>
48         </form>
49     </main>
50 </body>
51 </html>
```

Simple Login System III

welcome.php

```
1  #!/usr/local/bin/php
2  <?php
3      session_save_path(dirname(realpath(__FILE__)) . '/sessions/');
4      session_name('Demo'); // resume Demo session
5      session_start(); // start a session
6  ?>
7  <!DOCTYPE html>
8  <?php
9      // either no session or not logged in
10     if(!isset($_SESSION['loggedin']) or !$_SESSION['loggedin']) { ?>
11 <html>
12 <head>
13     <title>Unwelcome</title>
14 </head>
15 <body>
16     <p>Go back and log in.</p>
17 </body>
18 </html> <?php
19     }
20     else { // then they are logged in for real ?>
21 <html>
22 <head>
23     <title>Welcome</title>
24 </head>
25 <body>
26     <p>Welcome!</p>
27 </body>
28 </html> <?php
29     } ?>
```

Simple Login System

Some explanation...

Some servers block access to a temporary folder for sessions so we specify a session save path. The **dirname(realpath(__FILE__))** is a special way to obtain the full working directory of the *current PHP* script. In this case, session data will be stored in a **sessions** folder local to where the PHP script runs. We need to create such a folder first and set its permissions to 755.

At the start of **login.php**, we name the session '**Demo**' and start it. We set **\$_SESSION['loggedin']** to **false** because the visitor is not yet logged in, say. Or we could check if it is set and assign accordingly.

The **validate** function is defined to read the password from the file and compare a visitor's password to the true password. If they are equal then the session variable for being logged in is set to true and they are redirected to **welcome.php**. Otherwise the **\$error** flag is set.

Simple Login System

During the running of **login.php**, **\$error** is initially **false** and we check if there has been a **POST** submission. If not, we do nothing. If something was **POST**ed, then we run the validation.

The HTML includes a form and an error message if there was a **POST** and the **\$error** is flagged as **true**.

Simple Login System

For **welcome.php**, we again start the '**Demo**' session to access **\$_SESSION** and its variables.

If either **\$_SESSION['loggedin']** is not set (so it was never initialized) then clearly the user shouldn't be there. Or if **\$_SESSION['loggedin']** is **false**, they also shouldn't be welcomed. So we render HTML accordingly.

Otherwise, we welcome the visitor.

Simple Login System

When building more complicated pages where there can be multiple things requiring headers, which also includes the **header('Location: where_to_go')**, PHP can throw warnings that the headers have already been sent.

One fix is to make **ob_start()**; the first line of PHP code inside the PHP tags. This clears up previous headers and prevents headers from getting sent too early.

Passwords

Passwords should not be stored in their raw form. As a security measure, we can **hash** using a one-directional hashing function into something that looks like absolute gibberish. The general syntax to do so is:

```
$hashed_value = hash('method', $value);
```

where **method** can be one of many, many hashing algorithms. Using **'md5'** as one example:

```
// becomes: 5d41402abc4b2a76b9719d911017c592  
$hashed_hello = hash('md5', 'hello');
```

We never need to "invert" the mapping. If someone gives us a password, that can be hashed, too, and if it isn't the same as the desired hashed value of the correct password, their password is wrong.

Email

Using PHP we can send emails through the server with the **mail** function.

```
1  #!/usr/local/bin/php
2  <?php
3  // The message
4  $message = "This message is going to be sent in the email.\nThis is the second line of the
           email.";
5
6  // Send
7  mail('foo@bar.com', 'Subject Line', $message);
8  ?>
```

The **mail** function accepts an email address to send to, a subject, and a message body.

Some developers recommend preventing lines of an email from being more than 70 characters by adding:

`$message = wordwrap($message, 70, "\r\n");`

This inserts a return carriage and new line if a line is "too long". In modern email clients, this is seldom necessary.

HTML Forms

An HTML form can have both **action** and **onsubmit** attributes. When a form is submitted, the script from **onsubmit** is processed first, if anything is given, and it will return **true** if everything is okay and **false** to cancel the submission.

If it returns **true**, the user is redirected to the page specified by **action** with the appropriate method set.

By default, **onsubmit** returns **true** and **action** links back to the page.

HTML Forms I

Consider the files

index.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Onsubmit</title>
5    <script src="sub.js" defer</script>
6  </head>
7  <body>
8    <form action="other_page.html">
9      <input type="submit" value="submit" />
10   </form>
11   <form action="write.php" method="post" id="check_onsubmit">
12     <input type="text" id="x" name="x" />
13     <input type="submit" value="submit" />
14   </form>
15 </body>
16 </html>
```

sub.js

```
1  document.getElementById('check_onsubmit').onsubmit =
2    function() {
3      return document.getElementById('x').value.length !== 0;
4    };
```

write.php

HTML Forms II

```
1  #!/usr/local/bin/php
2  <?php
3      if(isset($_POST['x'])){ // check something posted
4          $file = fopen('x.txt', 'w');
5          fwrite($file, "$_POST['x']");
6          fclose($file);
7      }
8  ?>
```

HTML Forms

The **index.html** has 2 forms. The first form only has a submit button. Whenever it is pressed, the visitor is redirected to **other_page.html** due to the **action** specified.

The second form has a text input and a submit button. When someone tries to submit the form, JavaScript checks if they actually entered something. If they did not, it returns **false** but if they did, it returns **true** and the user is redirected from **action** to **write.php**.

In **write.php**, the **\$_POST** is active because the method in the form was **post** and we can write to a file here. Nothing displays on the **write.php** page.

It seems there is also an option to add a "submit" event listener but this does not have the same effect as editing the **onsubmit** property.

PHP Best Coding Practices

Having seen some of the basics of PHP, we can come up with a list of best practices. In addition to all the best coding practices for general programming and JavaScript, including commenting, function documentation, etc., we:

- ▶ Use `<?php ... ?>` instead of `<? ... ?>` for robustness.
- ▶ Avoid calling functions in loop conditions: initialize bounds outside of the function, for example.
- ▶ Use the `[]` to add a single element to an array rather than `array_push`.

PHP Best Coding Practices

To elaborate on a couple of those points:

// this is bad: count is called repeatedly and unnecessarily!

```
for($i = 0; $i < count($arr); ++$i) {  
    echo $arr[$i] , '<br/>';  
}
```

// this is good: only call count once

```
$len = count($arr);  
for($i = 0; $i < $len; ++$i) {  
    echo $arr[$i], '<br/>';  
}
```

Suppressing Warnings

Sometimes as programmers, we can manage possible failures ourselves but PHP still prints warnings/errors when things go awry.

```
1 <?php
2 $res = fopen('file_that_might_not_exist.txt', 'r');
3 if($res){ // worked
4     // ...
5 } else { // we KNOW it didn't work and have our own fix!
6     // do this
7 }
8 ?>
```

Above, despite our error management, PHP will still print a warning on the webpage if the file is not found. To silence warnings produced from a function call, we prefix the function name with @:

```
1 <?php
2 // no warnings will be printed
3 $res = @fopen('file_that_might_not_exist.txt', 'r');
4 ?>
```

Superglobals - \$_FILES

A web form that accepts files should have the tag structure:

```
<form enctype="multipart/form-data" method="post" ... >  
    ...  
</form>
```

The method should always be **post** for files and we specify data are encoded as "multipart/form-data".

Superglobals - \$_FILES

Once the form is submitted, the **\$_FILES** superglobal stores files that have been uploaded. Suppose we had an **<input type="file" name="paper" />** in our HTML form. We could then manage it with a PHP script:

```
1  $fileName = $_FILES['paper']['name'];
2  $saveLocation = dirname(realpath(__FILE__)) . '/uploads/' . $fileName;
3
4  move_uploaded_file($_FILES['paper']['tmp_name'], $saveLocation);
```

Each uploaded file results in a variable **\$_FILES['its_HTML_form_name']** with various properties such as **'name'**, the name the user gave to the file, and **'tmp_name'**, a temporary directory location given to the uploaded file.

Superglobals - \$_FILES

In specifying **\$saveLocation** we are saying we want to save the file to the **uploads** folder that lies within the current working directory of the PHP script. And the file name will be that of **\$fileName**.

The **move_uploaded_file** function transfers a file from its temporary location status to the desired directory.

Remark: this is kind of like the **mv** command in Linux that can both rename and move a file.

Superglobals - \$_FILES

The **uploads** folder should have permissions set to **755**.

Warning: to obtain the current folder path, do not use the **getcwd** PHP function as its behaviour varies across servers. On our servers, this function does not always return the current working directory, which is what it is supposed to return...

Superglobals - \$_FILES

Assuming the presence of a folder **uploads** with **755** permissions, the following PHP page allows a user to upload a file to that directory.

upload.php

```
1  #!/usr/local/bin/php
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <title>File Upload Demo</title>
6  </head>
7  <body>
8      <form action="<?php echo $_SERVER['PHP_SELF'];?>" enctype="multipart/form-data" method="
        post">
9          <input type="file" name="their_file" />
10         <input type="submit" value="Submit" name="submit" />
11     </form>
12 </body>
13 <?php
14     if( isset($_POST['submit']) ){ // they did submit
15         $fileName = $_FILES['their_file']['name'];
16         $saveLocation = dirname(realpath(__FILE__)) . '/uploads/' . $fileName;
17         move_uploaded_file($_FILES['their_file']['tmp_name'], $saveLocation);
18     }
19     else{
20         echo 'no file submitted';
21     }
22 >?
23 </html>
```

Superglobals - \$_FILES

It is also possible to allow multiple files to be uploaded at once. In this case, one needs to set the **multiple** boolean attribute and to give the name of the files an array name with []. For example:

```
<input type="file" name="uploads[]" multiple />
```

In this case, the **\$_FILES** superglobal will have the same keys as before when indexed at the key corresponding to the file input name, such as **name** and **tmp_name**, but the corresponding values at these keys will be an array corresponding to each file uploaded. For instance, **\$_FILES['uploads']** could be an array with a key of **name** with a value of an array storing 'file1.txt' and 'file2.txt'.

JSON

JavaScript Object Notation (JSON) is a means of encoding objects in JavaScript. This format is also conducive to sending information between JavaScript and PHP scripts. A JSON object must always be form the form:

```
{  
  "name": value,  
  "next_name": next_value  
}
```

In this case, all named variables appear in double quotes! The values can be numbers, booleans, strings, null, arrays (with []), or JSON objects.

JSON

Here is an example:

```
{  
  "name": "Joe Bruin",  
  "age": 20,  
  "program": {  
    "year": 3,  
    "specialization": "computing",  
    "courses": ["PIC 10A", "PIC 10B", "PIC 10C", "PIC 40A"]  
  }  
}
```

Note, despite the name "JavaScript Object Notation", this is not a valid JavaScript object, either, as the properties are given in double-quotes.

AJAX Calling PHP Scripts

An AJAX call can be made to a PHP script. Whatever the PHP script **echos** is what the AJAX call receives. In the following example, we will add a new customer to our clientele list that is stored in a flat file **clients.txt** with values **Company [TAB] Contact Name [TAB] Email Address**.

We will send the data through an AJAX call with POST; the data will be bundled in a **Client** object.

AJAX Calling PHP Scripts I

index.html

```
1  <!-- usual header stuff -->
2  <form>
3    <label for="company">Company:</label> <input type="text" name="company" id="company" />
      <br/>
4    <label for="contact">Contact Name:</label> <input type="text" name="contact" id="contact"
      /> <br/>
5    <label for="email">Email:</label> <input type="email" name="email" id="email" /> <br/>
6    <input type="button" value="Add Contact" id="add" />
7    <div id="confirm"></div>
8  </form>
9  <!-- more HTML -->
```

process.js

```
1  document.getElementById("add").addEventListener("click", add_client);
2
3  function Client(company, name, email){
4    this.company = company;
5    this.name = name;
6    this.email = email;
7  }
8
9
10 function add_client(){
11   const company = document.getElementById('company').value;
12   const name = document.getElementById('contact').value;
13   const email = document.getElementById('email').value;
14
15   const client = new Client(company, name, email);
```


AJAX Calling PHP Scripts II

```
16     const data = JSON.stringify(client);
17
18     const xmlhttp = new XMLHttpRequest();
19     xmlhttp.onreadystatechange = function() {
20         if (this.readyState === 4 && this.status === 200) {
21             document.getElementById('confirm').innerHTML = this.responseText;
22         }
23     };
24
25     xmlhttp.open("POST", "client_post.php", true);
26     xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
27     xmlhttp.send("client=" + data);
28 }
```

client_post.php

```
1  #!/usr/local/bin/php
2  <?php
3  // response type will be text
4  header('Content-Type: text/plain; charset=utf-8');
5
6  $client = json_decode($_POST['client'], false);
7
8  $project_space = 100;
9
10 $file = fopen('contacts.txt', 'a');
11 if($file){
12     fwrite($file, "{$client->company}\t{$client->name}\t{$client->email}\t");
13     for($i=0; $i<$project_space;++$i){ // add 100 spaces for 'extra space'
14         fwrite($file, ' ');
15     }
16 }
```

AJAX Calling PHP Scripts III

```
16     fwrite($file, "\n"); // go to new line
17     fclose($file);
18     echo 'written';
19 }
20 else{
21     echo 'file fail';
22 }
23 ?>
```

AJAX Calling PHP Scripts

Now we can unpack all of this. The boring, easy part is **index.html** that includes the **process.js** script and simply has a web form so that when the button is pressed, an AJAX call is made that will eventually modify the content of the **div** with ID of "confirm".

The **process.js** has a constructor for a **Client** class that makes an object to represent a client. The **add_client** button collects the information from the form, makes a **Client** and then turns that object into a JSON string by calling **JSON.stringify**.

AJAX Calling PHP Scripts

There is then making an AJAX call. As always, we make an **XMLHttpRequest** object and specify a callback to run when a response is received. In this case we wish to edit the **div**. This time, the **open** function specifies the method as "POST", the script to call as **client_post.php**, and that the call runs asynchronously (**true**), i.e., while the rest of the JS runs, the PHP can run.

Unlike our previous AJAX example, this time we have data to send so we specify what to send as a query. Recall queries are of the form **name1=value1&name2=value2....**

Remark: as a rule of thumb based on security and size limits of **GET**, AJAX should be called with **GET** for **safe** operations, i.e., operations that retrieve information but do not modify data on the server, and **POST** should be used whenever data are modified or information needs to be more secure.

AJAX Calling PHP Scripts

Within the PHP script, we use the **json_decode** function to convert the **\$_POST['client']** string into an object. The boolean argument is used to request an associative array if **true** instead of an object if **false**.

We then open a file and begin writing. Double quotes are used to evaluate the variables. The extra braces { ... } around the properties are used to group **\$client->company** as a single expression to be evaluated, for example. For objects in PHP, methods and properties are accessed via ->.

After writing to the file, the PHP script renders "written" or if there has been an error and the file pointer is **false** then it renders "file fail". Either message is then relayed as the value of **this.responseText** in the AJAX call.

AJAX Calling PHP Scripts

Remarks: HTTP specifies a protocol for how data are sent/received. Before calling **xmlhttp.send**, we specify what is being sent and in what form:

```
xmlhttp.setRequestHeader("Content-type",  
"application/x-www-form-urlencoded");
```

This sends a **header**, something storing meta-data to the PHP script that the way content is encoded is through a query string (like with GET).

Likewise, the PHP script adds a header (recall **headers** must be set before any HTML appears) to specify that it is returning plain text.

```
header("Content-Type: text/plain; charset=utf-8");
```

AJAX Calling PHP Scripts

Another remark: updating the flat file to add projects with a client, removing a client, etc. could all become very messy. PHP does support "random access" within a file just as C/C++ with **fseek** but nobody likes working that way...

Ultimately, using a database is a smarter/simpler solution, which will be covered later.

More on JSON

We have seen turning JavaScript objects into JSON with **JSON.stringify** and that PHP can decode JSON strings into arrays/objects with **json_decode**.

It goes the other way, too. We can convert a PHP object into a JSON string with **json_encode**. This will produce a string that in pure JS is interpreted as an array.

JavaScript can also parse JSON strings with **JSON.parse**.

Warning: it is imperative to *not* put a space between the variable encoding name, the equals sign, and its value. For example, a query string of form **x = 7** could fail catastrophically and leave **x** as **NULL** on the PHP end.

POSTing without a Submit Button I

It is possible to send a post request through AJAX calls, too, even without a form — at least not one the user interacts with.

In this example, the user can click on the a colour and be redirected to a PHP page with that colour as its background.

POSTing without a Submit Button II

Click on one of the words below:

RED

GREEN

Visitor can click on either word of a paragraph...



Visitor clicked red and was redirected.

POSTing without a Submit Button III

index.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <script src="redirect.js" defer></script>
5      <title>Colour Selector</title>
6  </head>
7  <body>
8      <p>Click on one of the words below:</p>
9      <p id="red">RED</p>
10     <p id="green">GREEN</p>
11 </body>
12 </html>
```

POSTing without a Submit Button IV

The JavaScript creates a form (which the user never sees) and creates an element **<input type="hidden" ... />**, which can be used on all forms to add additional data that do not appear in the form. By calling **submit** on the form *once added to the document*, the form is submitted.

redirect.js

```
1  function redirect(colour){
2    // form to use post to redirect user
3    const form = document.createElement("form");
4    form.method = "post";
5    form.action = "display.php";
6
7    // add data to a hidden element
8    const data = document.createElement("input");
9    data.type = "hidden";
10   data.value = colour;
11   data.name = "colour";
12   form.appendChild(data);
13
14   // add to body and submit
15   const b = document.getElementsByTagName("body")[0];
16   b.appendChild(form);
17   form.submit();
18 }
19
20 document.getElementById("red").addEventListener("click",
```

POSTing without a Submit Button V

```
21     ()=> { redirect("red"); } );  
22  
23 document.getElementById("green").addEventListener( "click",  
24     ()=> { redirect("green"); } );
```

POSTing without a Submit Button VI

red_green.css

```
1  .red{
2      background-color: red;
3  }
4
5  .green{
6      background-color: green;
7  }
```

display.php

```
1  #!/usr/local/bin/php
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <title>Display</title>
6      <link href="red_green.css" rel="stylesheet" />
7  </head>
8  <body class= "<?php echo $_POST['colour']; ?>" >
9  </body>
10 </html>
```

Object Oriented Design I

We will give the "equivalent" PHP implementation to the C++ classes below:

```
class Square{
private:
    double length;

protected:
    double get_length() const { return length; }

public:
    Square(double _length) : length(_length) {} // (A)

    double get_area() const {return length*length;} // (B)
};

class ColouredSquare : public Square{ // (C)
```

Object Oriented Design II

```
private:
    std::string colour;

public:
    ColouredSquare(double _length, std::string _colour):
        Square(_length), colour(std::move(_colour)) {} // (D)

    void set_colour(std::string _new_colour) {
        colour = std::move(_new_colour);
    }

    void display() const { // (E)
        std::cout << colour + " square of length " <<
            get_length() << " has area" << get_area()
            << '\n';
    }
};
```


Object Oriented Design I

```
1  class Square{
2      private $length;
3
4      protected function get_length() { return $this->length; }
5
6      function __construct($length) { $this->length = $length; }
7
8      public function get_area() { return $this->length*$this->length; }
9  }
10
11 class ColouredSquare extends Square{
12     private $colour;
13
14     public function __construct($length, $colour){
15         parent::__construct($length);
16         $this->colour = $colour;
17     }
18
19     public function set_colour($new_colour){
20         $this->colour = $new_colour;
21     }
22
23     public function display(){
24         echo "{$this->colour} square of length {$this->get_length()} has area {$this->get_area()}
25             \n";
26     }
27 }
```

Access Modifiers

Like C++, PHP supports **public**, **protected**, and **private** access modifiers. In PHP, they appear right before the method or property. Without listing a modifier, the default behaviour is **public**.

All properties (member variables) must be prefixed by something. If not **public**, **protected**, or **private**, then the **var** keyword granting public access must be used.

Access Modifiers

Anything that is **public** is accessible to all users of the class and all derived classes. Anything that is **protected** is only accessible within the class or its derived classes. Anything that is **private** is only accessible within the class itself.

Remark: **protected** should only ever be used for member functions. A property should never be **protected**. Ideally **protected** shouldn't be used at all.

Constructors and \$this

A class constructor (used to make a class object) is a function that must always be named **__construct**. Destructors (what takes place as an object is destroyed) can also be written and they have the name **__destruct**.

The **\$this** variable is a reference to the class itself. To access members (properties or methods) of the class, we must use **\$this->**.

When accessing a property, we don't need to prefix that property with a \$ again.

Like other functions in PHP, overloading is not possible.

Inheritance

To inherit the properties and methods of another class, PHP uses the **extends** keyword. Hence, **ColouredSquare extends Square**.

To invoke the constructor of the base class, we scope it with **parent::**.

Function Overriding

A derived class can overwrite how a method is implemented. We will write the PHP equivalent of the C++ below:

```
struct Foo{  
    virtual void print() const { std::cout << "Foo!"; }  
};  
  
struct Bar : Foo{  
    void print() const override { std::cout << "Bar!"; }  
};  
  
// in code  
Foo f; f.print(); // prints Foo!  
Bar b; b.print(); // prints Bar!
```

Function Overriding

In PHP:

```
1  class Foo{
2      public function print() { echo 'Foo!'; }
3  }
4
5  class Bar extends Foo{
6      public function print() { echo 'Bar!'; }
7  }
8
9  # now we use these
10 $f = new Foo;
11 $f->print(); # prints Foo!
12
13 $b = new Bar;
14 $b->print(); # prints Bar!
```

Remark: if no constructor is provided, PHP supplies a “default constructor” for the classes making all values NULL. When the **new** expression is used and the class is given no input arguments, it gets default constructed.

Polymorphism

Unlike C++, PHP will treat function arguments, whether or not they are passed as references/pointers **polymorphically**. The term **polymorphism** in programming refers to being able to handle multiple kinds of objects with a single underlying object type.

We can write a function **manage_Foos** that will only accept objects of type **Foo** or those derived from it.

```
1  function manage_Foos (Foo $a_foo) {  
2      $a_foo->print();  
3  }  
4  
5  manage_Foos($f); // prints Foo!  
6  manage_Foos($b); // prints Bar!  
7  manage_Foos(7); // ERROR
```

For classes, PHP allows us to specify an input data type for a function.

Object Oriented Design

PHP also has **abstract** classes and **interfaces**. As in other languages, they specify what a derived class should do/implement. We won't go into that.

Unlike C++, PHP does not support multiple inheritance. A class **A** cannot extend both **B** and **C**.

Objects Passed by Reference

In PHP, an object that has been **newed** is effectively a pointer to the location in memory where the real object lies. Because of this, objects are passed by reference, even if their members are mutated! To force a copy, we use the **clone** keyword (makes a shallow copy) or **deep_copy** function (forces a deep copy).

zval

PHP is written in C. The handling of the various data types is done through a small collection of classes to store type information and values. Understanding how these classes work together helps understanding more about PHP's various optimizations and behaviours.

The key players that we will briefly examine include:

- ▶ **zval**, a class that stores a value and a flag for what type of data it actually is;
- ▶ **zend_value**, a class that stores members appropriate for various data types;
- ▶ **zend_reference**, a class that represents a reference type; and
- ▶ others such as **zend_object**, **zend_array**, and **zend_string** to represent objects, arrays, strings, and so forth.

We will examine an *approximation* to these classes in C++. Various aspects of the classes are suppressed/modified to make for simpler exposition.

zval

```
struct zval {  
    zend_value val;  
    unsigned int type;  
};
```

The actual value, **val**, of the **zval** is stored in the **zend_value** member. But the **zend_value** member is so basic, it doesn't even know what it is. So there are integer flags for the various **types**, e.g.

- ▶ IS_LONG == 4
- ▶ IS_DOUBLE == 5
- ▶ IS_STRING == 6
- ▶ IS_ARRAY == 7
- ▶ IS_OBJECT == 8
- ▶ IS_REFERENCE == 10

zval

```
union zend_value {  
    long lval;  
    double dval;  
    zend_string *str;  
    zend_array *arr;  
    zend_object *obj;  
    zend_reference *ref;  
    // ...  
};
```

A **union** is a class type where only one member variable at a time is active. All other values are in an undefined state. The size of a union is the size of its largest member so lots of different data types can all fit together under one roof.

Types other than the simple integers or floating points are acquired on the heap and managed through pointers to the relevant types.

zval

```
struct zend_reference {  
    unsigned int ref_count;  
    zval val;  
    // ...  
};
```

```
struct zend_object {  
    unsigned int ref_count;  
    // other pointers and such  
};
```

The reference, object, array, and string types all have similar interfaces. They need to be freed properly when nothing can reference them anymore, thus the need for a reference count.

A reference type internally stores a **zval** for what it is referencing.

zval

Integers and floating points are stored directly in their **zvals** without the indirection of pointers.

When a variable **\$b** is initialized from or created through *requesting a reference* of a variable **\$a**, i.e., through **&\$a**, *both \$b and \$a* become reference types.

The **copy on write** is applied through these **zvals**. Until a write is done, variables can share a pointer value to a common array/string buffer, etc. The copy on write does not happen with objects.

PHP may do additional optimizations that seem to violate these rules for arrays and strings.

zval

We consider working with some numbers first.

```
$a=0;
```

```
zval_a  
type=long  
value:0
```


zval

We consider working with some numbers first.

```
$a=0;
```

```
$b=$a;
```

```
zval_a  
  type=long  
  value:0
```


```
zval_b  
  type=long  
  value:0
```


zval

We consider working with some numbers first.

```
$a=0;  
$b=$a;  
$c=&$a;
```

zval_a
type=reference
value: 

zval_b
type=long
value: 

zval_c
type=reference
value: 

ref_ac
count: 2
value:

zval:
type=long
value: 0



zval

Here's what happens with arrays. Note the copy on write.

```
$a=[1];
```

```
$a=[1]; zval_a
type=array
value: 1
```



```
arr_a
count=1
value: [1]
```

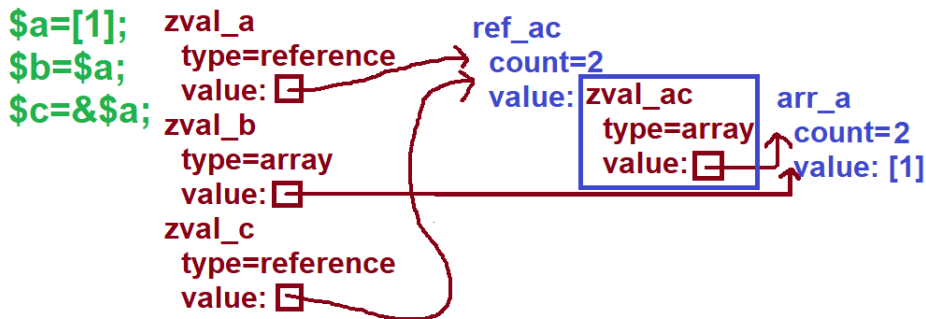
zval

Here's what happens with arrays. Note the copy on write.



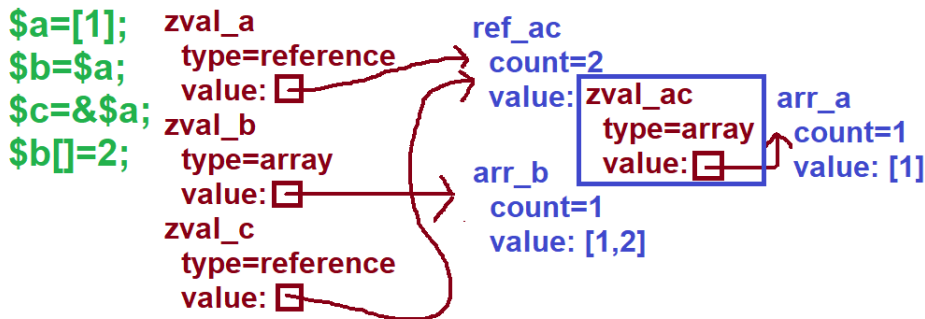
zval

Here's what happens with arrays. Note the copy on write.



zval


Here's what happens with arrays. Note the copy on write.



zval

And here's an illustration for objects.

```
class Foo{  
    var $i = 0;  
}
```

zval_a
type= object
value: 

obj_a
count=1
value: {i=0}

```
$a=new Foo();
```

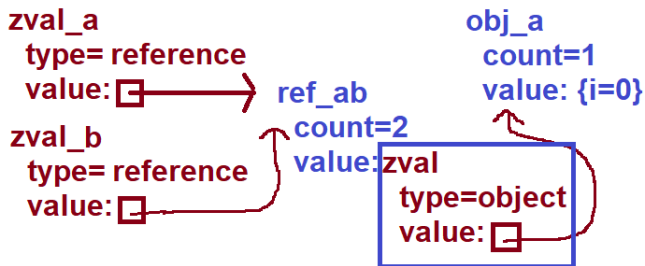


zval

And here's an illustration for objects.

```
class Foo{  
    var $i = 0;  
}
```

```
$a=new Foo();  
$b=&$a;
```

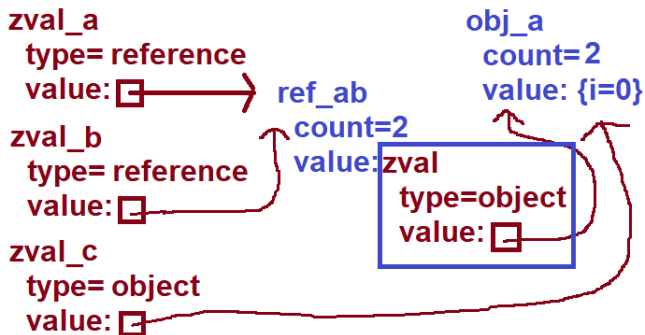


zval

And here's an illustration for objects.

```
class Foo{  
    var $i = 0;  
}
```

```
$a=new Foo();  
$b=&$a;  
$c=$a;
```

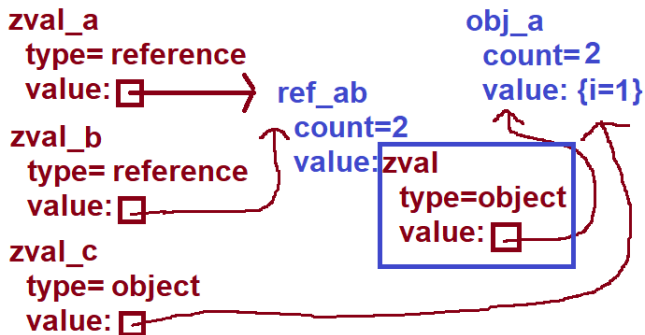


zval

And here's an illustration for objects.

```
class Foo{  
    var $i = 0;  
}
```

```
$a=new Foo();  
$b=&$a;  
$c=$a;  
$c->i=1;
```

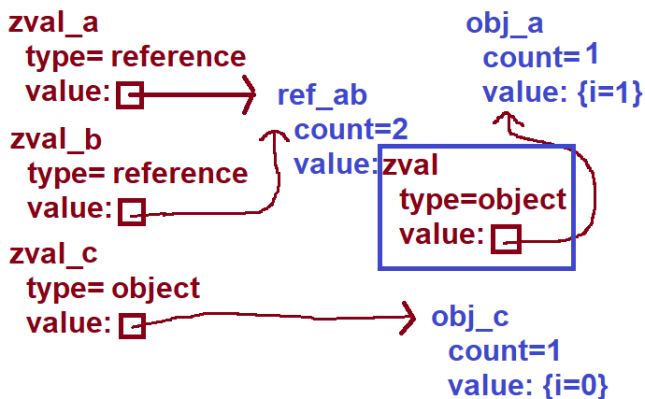


zval

And here's an illustration for objects.

```
class Foo{  
    var $i = 0;  
}
```

```
$a=new Foo();  
$b=&$a;  
$c=$a;  
$c->i=1;  
$c=new Foo();
```

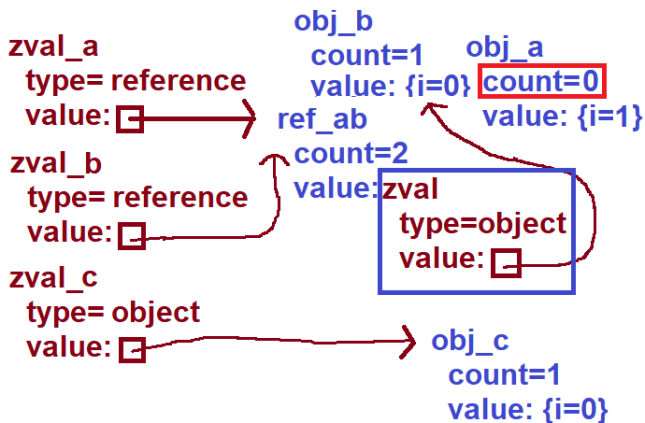


zval

And here's an illustration for objects.

```
class Foo{  
    var $i = 0;  
}
```

```
$a=new Foo();  
$b=&$a;  
$c=$a;  
$c->i=1;  
$c=new Foo();  
$b=new Foo();
```



var_dump and debug_zval_dump

There is also a quirky function called **debug_zval_dump** which is difficult to use since its implementation has changed a lot over different PHP versions and the answers it outputs depend on optimizations of the PHP engine.

Nonetheless, it does illustrate the concept/idea that the variables are represented as **zvals**.

```
$x = 'hello';  
$y = $x;
```

```
debug_zval_dump($x); // string(5) "hello" refcount(3)
```

The comment there is the output in PHP 5. In PHP 7, the refcount is 1.

Garbage Collection

PHP uses reference counting to determine when an object should be destructured.

PHP 5 introduces a destructor concept similar to that of other object-oriented languages, such as C++. The destructor method will be called as soon as there are no other references to a particular object, or in any order during the shutdown sequence. ~PHP Net Manual

Thus, for example, if we define some variables in a function, upon exiting the function, as long as each variable is unreferenced after the function ends, they will be destroyed.

In effect, PHP variables behave like objects referenced through smart pointers in C++, automatically adhering to RAI, provided there's no funny circular references going on.

Memory Leaks

In some cases, since the garbage collector does reference counting, memory leaks can arise!

```
1  class Person{
2      private $name;
3      private $friend;
4
5      function __construct($name) {
6          $this->name = $name;
7          $this->friend = NULL;
8      }
9
10     function __destruct() {
11         echo "destroying {$this->name}<br/>";
12     }
13
14     function set($friend){
15         $this->friend = $friend;
16     }
17 }
18
19 function Leaks(){
20     $alice = new Person('Alice');
21     $bob = new Person('Bob');
22     $alice->set($bob);
23     $bob->set($alice);
24     $colleen = new Person('Colleen'); // has no friends
25 }
26
27 Leaks();
28
29 echo 'function call has ended', '<br/>';
```

Memory Leaks

This generates the output:

```
destroying Colleen  
function call has ended  
destroying Alice  
destroying Bob
```

Alice and Bob are only destroyed at the end of the program, not the end of the function. There is a memory leak.

Since a PHP programmer does not have control over memory management, it is the programmer's responsibility to avoid situations where memory leaks could arise.

Fun remark: Python also has this same flaw and will leak memory in the same situation.

Copy on Write for Arrays

For arrays, it seems that the moment any level of modification takes place within the array, no matter how deep, the array is copied.

```
1  <?php
2  class Foo{
3      var $x;
4      function __construct($i) { $this->x = $i;}
5  }
6
7  $u = new Foo(4);
8  $a = [new Foo(3), &$u];
9  $b = $a; // $b and $a both reference the same array
10 $c = $a; // $c and $a both reference the same array
11 $b[0]->x = 1; // now $b is no longer referencing what $a does
12 $c[1]->x = 1; // now $c is no longer referencing what $a does
13
14 $d = new Foo( new Foo(0) );
15 $e = $d; // $e references what $d does
16 $e->x->x = 30; // $e still references what $d does as $e->x ``pointer" unchanged
17 $e->x = new Foo(1); // $e and $d still reference the same object
18 $e = new Foo(1); // $e and $d reference different objects
19 ?>
```

Warning: it is important not to get **PHP References** (with the **&**) mixed up with the notion of a variable referencing an object or other entity.

Returning References

To return a reference from a function, we prepend the function name with an **&**:

```
function &add(&$x) {  
    ++$x;  
    return $x;  
}
```

```
$y = 10;  
add(add($y)); // now $y is 12
```

Cloning I

To illustrate the **clone** feature, we consider a class to model an item and its price. We note the difference in behaviour between when **clone** is/is not used.

```
1  <?php
2  class item{
3      private $price;
4      private $name;
5
6      public function __construct($p,$n){
7          $this->price = $p;
8          $this->name = $n;
9      }
10
11     public function get_name(){
12         return $this->name;
13     }
14
15     public function get_price(){
16         return $this->price;
17     }
18
19     public function give_discount($d){
20         $this->price -= $d;
21     }
22
23     public function __clone(){
24         return new item($this->price, $this->name);
25     }
26 }
```

Cloning II

```
27
28 $salad = new item(5.29, "salad");
29 $brownie = new item(6.00, "brownie");
30
31 // not independent copy
32 $brownie2 = $brownie;
33
34 // makes a
35 $brownie3 = clone $brownie;
36
37 $brownie->give_discount(1);
38
39 echo $brownie->get_price(), '<br/>'; // 5
40 echo $brownie2->get_price(), '<br/>'; // 5
41 echo $brownie3->get_price(), '<br/>'; // 6
42 ?>
```

Remark: if an object being **cloned** itself has member variables that are objects and those objects do not have a specialized **clone** function written, the clone and the original will share the same pointer/reference to such members.

Traits I

In PHP, **traits** allow for inheritance-like behaviour for classes that may not be part of the same inheritance hierarchy. In other languages, these may be called **mixins** where a “base” class adds special methods to “derived” classes.

```
1  <?php
2  trait tax{
3      var $t = 1.1; // tax rate
4      function with_tax() {
5          return $this->price*$this->t;
6      }
7  }
8
9
10
11 class item{
12     use tax; // include trait functionality
13
14     private $price;
15     private $name;
16
17     // uses the trait
18     public function get_price_with_tax(){
19         return $this->with_tax();
20     }
21
22     /* rest as before */
23 }
```

Traits II

```
24
25 $brownie = new item(6.00, "brownie");
26
27 echo $brownie->get_price_with_tax(); // 6.6
28 ?>
```

PHP Injecting Code into JavaScript

Here is a short example of PHP injecting code into JS.

```
1  #!/usr/local/bin/php
2  <?php
3      class Person{
4          var $name;
5          var $age;
6          public function __construct($name, $age){
7              $this->name = $name;
8              $this->age = $age;
9          }
10     }
11     $michelle = new Person('Michelle', 60); // make PHP object
12     $php_json_string = json_encode($michelle); // turn to JSON string
13     ?>
14     <!DOCTYPE html>
15     <head>
16         <title>PHP to JS</title>
17     </head>
18     </body>
19     <input type="button" value="Show Info" onclick="alert(michelle.name + ' is ' +
        michelle.age + ' years old.');" />
20     <script> <!-- direct JS -->
21         // Parse the output as a string
22         let michelle = JSON.parse('<?php echo $php_json_string; ?>');
23         // or could just view as an array directly
24         // let michelle = <?php echo $php_json_string; ?>;
25     </script>
26 </body>
27 </html>
```

PHP Injecting Code into JavaScript

The PHP makes all the data. It also renders some **script** tags where in JS, the **michelle** variable is defined through the rendering of PHP.

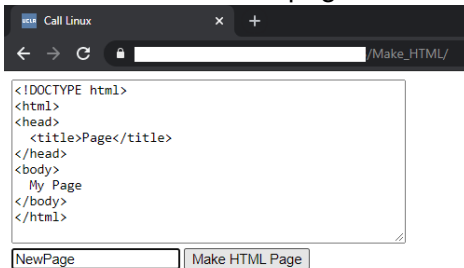
The pure **\$php_json_string** when viewed as JS is an array. But when enclosed in quotes, it is a JSON string to be parsed.

When the button is pressed, we see the alert, "Michelle is 60 years old."

Remark: in some cases such as where PHP needs to make JS, it can be okay to have JS directly on the page. The same goes for when PHP needs to render CSS, which it sometimes does: in these less common cases, CSS may also be defined on the page.

PHP Calling Linux Commands

Through PHP, we can run Linux commands. We will look at a sort of meta example: we will write a web page where the user can enter HTML in a text area and give a directory name. When they submit their data, a new directory is created/overwritten with the name they prescribed and given the file **index.html** storing their HTML. They also then get redirected to this new page.

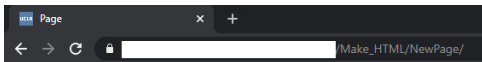


Call Linux

← → ↻ 🔒 /Make_HTML/

```
<!DOCTYPE html>
<html>
<head>
  <title>Page</title>
</head>
<body>
  My Page
</body>
</html>
```

NewPage Make HTML Page



PHP Calling Linux Commands

The main command to be aware of is the **exec** command that runs Linux commands.

Also: **mkdir -p Name** will create a directory named “Name” if it does not already exist. In general an error can be thrown if one tries to create a directory that already exists.

PHP Calling Linux Commands I

index.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Call Linux</title>
5    <script src="make.js" defer></script>
6  </head>
7  <body>
8    <textarea rows="10" cols="50" id="html" name="html"></textarea>
9    <br/>
10   <input type="text" id="folder" name="folder" />
11   <input type="button" id="make" value="Make HTML Page" />
12 </body>
13 </html>
```

make.js

```
1  document.getElementById('make').addEventListener("click",
2    function() {
3      const x = new XMLHttpRequest();
4      x.onreadystatechange = function(){
5        // when things are a go, we simply redirect
6        if( this.status === 200 && this.readyState == 4){
7          const folder = document.getElementById('folder').value;
8          window.location = "." + folder;
9        }
10     }
11     x.open("POST", "make.php", true); // send through post
12     x.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
13
```

PHP Calling Linux Commands II

```
14     const data = {
15         folder: document.getElementById('folder').value,
16         content: document.getElementById('html').value
17     };
18
19     x.send("data=" + JSON.stringify(data));
20 }
21 );
```

make.php

```
1  #!/usr/local/bin/php
2  <?php
3      // decode into object
4      $data = json_decode($_POST['data'], false);
5
6      $folder = $data->folder;
7      $content = $data->content;
8
9      // try to make the folder
10     exec("mkdir -p $folder");
11
12     // open the index.html file and write
13     $file = fopen("./$folder/index.html", 'w');
14     fwrite($file, $content);
15     fclose($file);
16
17     // change permissions within the new folder
18     exec("chmod 755 $folder/*");
19     ?>
```

Cross Origin Support

By default, AJAX requests originating from other servers are blocked. Thus, if we have data we want to be made available on domain X and JavaScript on domain Y requests it, the request cannot be processed.

By enabling **Cross Origin Support (CORS)**, we can allow this functionality. This is done in PHP with setting a header property:

```
header("Access-Control-Allow-Origin: *");
```

or

```
header("Access-Control-Allow-Origin: specific_domain");
```

The * allows for all domains. But we can also specify a specific domain.

Cross Origin Support

give_num.php at http://www.example1.com/get_num.php

```
1  #!/usr/local/bin/php
2  <?php
3      header("Access-Control-Allow-Origin: http://www.example2.com");
4      echo 42;
5  ?>
```

seek_num.js at <http://www.example2.com>

```
1  function get_num(){
2
3      let xmlhttp = new XMLHttpRequest();
4      xmlhttp.onreadystatechange = function() {
5          if (this.readyState === 4 && this.status === 200) {
6              alert(this.responseText);
7          }
8      };
9
10     xmlhttp.open("GET", "http://www.example1.com/get\_num.php", true);
11     xmlhttp.send();
12 }
```

index.html at <http://www.example2.com>

```
1  <!DOCTYPE html>
2  <head>
3      <title>CORS Demo</title>
4      <script src="seek_num.js" defer></script>
5  </head>
6  <body></body>
```

HTTP Methods

The HTTP protocol supports **GET**, **POST**, **PUT**, and **DELETE** methods. In understanding these methods, we must recognize whether an operation is **safe** (does not modify data on the server) and whether an operation is **idempotent** (when an identical request is made, there is no change on the server).

- ▶ **GET** is safe and idempotent (just to retrieve information).
- ▶ **POST** is not safe and is not idempotent (possibly creating a file or duplicate of something).
- ▶ **PUT** is not safe but is idempotent (posting something into a given file, say).
- ▶ **DELETE** is not safe but is idempotent (removing a file, say).

Additionally, **PUT** and **DELETE** are not supported as **methods** in HTML forms: they need to be done with AJAX.

PUT and DELETE

Here we consider a simple example of **PUT** and **DELETE**. We provide an interface where a user can **touch** to create a file or **rm** to remove it.

file name:

touch

rm

Repeating a **touch** or **rm** has no observable effect (well, technically the file will have a new time stamp if **touched** again).

PUT and DELETE I

index.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>PUT/DELETE</title>
5    <script src="put_delete.js?v=0" defer></script>
6  </head>
7  <body>
8    <form>
9      <label for="file">file name: </label><input type="text" id="file" name="file" />
10     <br/>
11     <input type="button" id="put_button" value="touch" /> <input type="button" id="
      delete_button" value="rm" />
12   </form>
13 </body>
14 </html>
```

put_delete.js

PUT and DELETE II

```
1  function do_ajax(method, element){
2      const file = element.value;
3
4      const x = new XMLHttpRequest();
5
6      x.onreadystatechange = function(){
7          if(this.status === 200 && this.readyState === 4){
8              alert("done");
9          }
10     };
11
12     x.open(method, "action.php", true);
13     x.setRequestHeader("application-type", "application/json");
14
15     const data = {
16         fname: file
17     };
18
19     x.send( "data="+JSON.stringify(data) );
20 }
21
22 // this function returns a function!
23 // the function returned will invoke either put/delete functions
24 function make_method(method){
25     return ()=>{ do_ajax(method, document.getElementById("file") ); };
26 }
27
28 document.getElementById('put_button').addEventListener("click",
29     make_method("put") );
30
31 document.getElementById('delete_button').addEventListener("click",
32     make_method("delete") );
```

PUT and DELETE III

action.php

```
1  #!/usr/local/bin/php
2  <?php
3  header('content-type: text/plain');
4  // parse the input into an array called $input
5  parse_str(file_get_contents('php://input'), $input );
6
7  $as_json = $input['data']; // read 'data' property of array
8  $as_object = json_decode($as_json, false); // is JSON string to parse
9
10 $file = $as_object->fname; // extract the file object
11
12 // guard against deleting the essential files!
13 if($file === 'index.html' || $file === 'action.php' || $file === 'put_delete.js' ){
14     exit;
15 }
16
17 $last_slash = strrpos($file, '/'); // find position of last slash, if any
18 if($last_slash !== false){ // so was found
19     if($last_slash === strlen($file)-1 ){ // so no file
20         exit;
21     }
22     $file = substr($file, $last_slash+1);
23 }
24
25 if($_SERVER['REQUEST_METHOD'] === 'PUT'){ // with put we make file
26     exec("touch $file");
27 }
28 else if ($_SERVER['REQUEST_METHOD'] === 'DELETE'){ // with delete, remove it
29     exec("rm -f $file");
30 }
```

PUT and DELETE IV

31 ?>

The AJAX call syntax is the same except that we use either the **put** or **get** methods.

Note how the event listeners are set by calling a function that itself returns a function. This isn't required but is kind of nice.

There are no **\$_PUT** or **\$_DELETE** superglobals. When inputs are passed to PHP scripts, they can be accessed through a file **php://input**. This file can only be read once. **file_get_contents** retrieves the contents of a file which we, through **parse_str**, write to an array called **\$input** (or any name we wanted).

Since the methods were sent with a **data=value** query, we retrieve a JSON string of the data through accessing the 'data' property and then

PUT and DELETE V

decode that to an object that will have “fname” for the file name (“fname” was the name of the property we gave in JS).

By examining **`$_SERVER['REQUEST_METHOD']`** we can determine what was called.

We parse the filename string in case it links to higher level directories, etc., by only considering the part of the name after its last slash (found with **`strrpos`**). We need to be careful not to allow **`false`** to be coerced to **`0`**.

PHP Downloads I

In this example, we'll consider a mechanisms by which a user can upload a file of **.jpg** extension and receive a download of the file converted to **.png**.

convert.jpg

user uploads file

No file chosen

 convert.png ^

converted file downloads

PHP Downloads II

On the main page, **index.php**, the user can upload a file. When they submit the file, it is uploaded to the server and JavaScript is injected to make an AJAX call.

We make use of the **mogrify** program included in many Linux systems. The syntax is:

mogrify -format [desired extension type of output] [file to transform].

index.php

```
1  #!/usr/local/bin/php
2  <!DOCTYPE html>
3  <html>
4  <head>
5      <title>JPG to PNG</title>
6      <script src="dl.js" defer></script>
7  </head>
8  <body>
9      <form action="<?php echo $_SERVER['PHP_SELF'];?>" enctype="multipart/form-data" method="
10         post">
11         <input type="file" name="their_file" />
12         <input type="submit" value="Submit" name="submit" />
13     </form>
14     <?php
15         if( isset($_POST['submit']) ){ // they did submit
16             $fileName = $_FILES['their_file']['name'];
17             $end = strrpos($fileName, ".jpg");
```

PHP Downloads III

```
17 $ext_len = 4;
18 if( ($end !== false) && ($end === strlen($fileName)-$ext_len) ){
19     $fileBase = substr($fileName, 0, strlen($fileName)-$ext_len);
20     $saveLocation = dirname(realpath(__FILE__)) . '/uploads/' . $fileName;
21     move_uploaded_file($_FILES['their_file']['tmp_name'], $saveLocation);
22     exec("mogrify -format png $saveLocation");
23     // inject call to AJAX
24     echo "<script>window.addEventListener('load', ()=>{ transform('$fileBase'); }) ;</script>";
25 }
26 else{
27     echo 'invalid file';
28 }
29 }
30 ?>
31 </body>
32 </html>
```

PHP Downloads IV

The JavaScript function in **dl.js** works with two separate PHP scripts: **check_file** which returns true only when/if the file has been successfully transformed within an allotted time and **download.php** that causes a file download.

dl.js

```
1  function transform(file){
2      const x = new XMLHttpRequest();
3      x.onreadystatechange = function(){
4          if(this.status===200 && this.readyState === 4){
5              if(this.responseText === "1"){ // if successful
6                  window.location = "download.php?file=" + file;
7              }
8          }
9      };
10     x.open("GET", "check_file.php?file="+file, true);
11     x.send();
12 }
```

PHP Downloads V

In **check_file.php**, we use the **file_exists** function that checks if a file of a given path exists. We also use **microtime** that, when fed a value **true**, returns a floating point representation of the number of seconds (down to microsecond precision) elapsed microseconds since January 1, 1970, 00:00 UTC time.

check_file.php

```
1  #!/usr/local/bin/php
2  <?php
3      $fileName = "./uploads/" . $_GET['file'] . ".png";
4      $max_time = 1.; // do not wait longer than 1 second
5      $start = microtime(true);
6      while( !file_exists($fileName) && microtime(true) - $start < $max_time){
7          ; // do nothing...
8      }
9      echo (int) file_exists($fileName); // could return 1 or 0
10     exit();
11     ?>
```

PHP Downloads VI

The **download.php** page actually prompts a download in the window whose header is sent there.

With specifying **Content-Type: application/octet-stream** in the header, the web browser understands the data type it receives is unknown and could be anything. With **Content-Disposition: attachment** in the header, the browser will interpret the data as an attachment to download. We then also specify the **filename=** to specify what file to download.

With **readfile**, the contents of a file are returned (and in this case downloaded).

PHP Downloads VII

download.php

```
1  #!/usr/local/bin/php
2  <?php
3      ob_start();
4      $outputName = $_GET['file'] . ".png";
5      $file = "./uploads/" . $outputName;
6      header("Content-Type: application/octet-stream");
7      header('Content-Disposition: attachment; filename="' . $outputName . '"');
8      readfile ($file);
9      exit();
10  ?>
```

Date

PHP has a **date** function that can be useful to extract the day/time information. In essence, it accepts a string and converts various letters of the string into values like the number of minutes of the current time, day of the month, etc.

Through the **date_default_timezone_set** function, we can set the timezone to be used: the default is the timezone of the server. A few valid time zones include:

America/Los_Angeles (Pacific)

America/Winnipeg (Central)

America/Toronto (Eastern)

(basically most major cities in the Americas can be used for calibrating the timezone)

Date

Before looking at an example, here are a few:

- ▶ **Y**: the full year
- ▶ **y**: the padded last two digits of the year from 00 to 99
- ▶ **M**: the abbreviated month
- ▶ **m**: the two-digit padded month number of the year from 01 to 12
- ▶ **D**: the abbreviated weekday
- ▶ **d**: the two-digit day of the month from 01 to 28/31
- ▶ **H**: the two-digit padded hour of the day from 00 to 23
- ▶ **h**: the two-digit padded hour of the day from 01 to 12 ignoring am/pm
- ▶ **i**: the two-digit padded number of minutes past the hour from 00 to 59
- ▶ **s**: the two-digit padded number of seconds past the minute from 00 to 59
- ▶ **A/a**: uppercase/lowercase ante or post merdien
- ▶ **T**: the timezone abbreviation

Date

Here is an example:

```
1  #!/usr/local/bin/php
2  <?php
3      $now = 'It is now ' . date('D, M d, Y') . ' at ' . date('h:ia--T');
4
5      date_default_timezone_set('America/Chicago');
6
7      $chicago = 'In Chicago, it is ' . date('d/m/y') . ' at ' . date('H:i:s--T');
8
9      echo $now, '<br/>', $chicago;
10  ?>
```

It is now Wed, May 24, 2022 at 11:12pm-PST
In Chicago, it is 24/05/22 at 13:12:05-CDT

Strict Types

Modern PHP allows for **strict typing**. This can be accomplished by adding **declare(strict_types = 1);** to a file. We can then specify the value a function returns by inserting **: type** before its body to enforce that **type** is returned. And similar to C++, we can also specify required input types — but **int** can be converted to **float**.

```
1  #!/usr/local/bin/php
2  <?php
3      declare(strict_types = 1);
4
5      function int_round(float $x): int {
6          if($x > 0):
7              return (int) ($x+0.5); // cast from float to int required!
8          else:
9              return (int) ($x-0.5); // cast required!
10         endif;
11     }
12
13     echo int_round(9.8); // okay, returns int 10
14     # echo int_round('12.34'); // ERROR: given string
15     echo int_round(4); // will be okay: int to float is allowed
16  ?>
```

Strict Types

When strict types are enabled, we can also write functions that do not enforce typing. By omitting the data type in the function argument or by omitting the **: return_type**, we allow any inputs and/or outputs of any type.

Summary

- ▶ PHP is a high level, interpreted programming language like JavaScript, which has loose typing and automatic memory management.
- ▶ PHP works to run code on the server side and the code is hidden from the visitors.
- ▶ For PHP to run on a server, one must link to the binary at the top **#!/usr/local/bin/php** — which could vary across servers.
- ▶ PHP has four scalar types, **boolean**, **integer**, **float**, and **string** and also has **arrays**, **objects**, **NULL**, and **resources**.
- ▶ All PHP variables, with the exception of **consts**, must be prefixed by a \$.
- ▶ Unlike JS, '+' will always mean addition (after coercions) and instead '.' is used for concatenation and will always convert its arguments to strings.
- ▶ In PHP single quotes '...' involve minimal parsing whereas double quotes "..." allow for string interpolation of variables and format sequences like new lines.

Summary

- ▶ PHP implements many of its functions as free functions. For example **substr** takes the string it is extracting a substring of as its first argument.
- ▶ PHP functions can sometimes return different data types!
- ▶ We can clean up a variable by **unset**ting it. But if this is done to an array, the corresponding index will be missing.
- ▶ Functions can either be defined by name or be anonymous functions.
- ▶ Anonymous functions, to form a closure, must declare the closure with a **use** expression.
- ▶ PHP admits alternate control flow structure with colons.
- ▶ Values defined in control flow will leak to the outside, function or global scope.
- ▶ Within a function, global variables are not accessible
- ▶ As in JS, objects should be viewed as pointers.
- ▶ In PHP, an **&** can be used to pass arguments or return by reference.
- ▶ By default, functions accept argument by reference but make a copy when the referenced variable would be written to.

Summary

- ▶ PHP has many super global variables like **\$_GET**, **\$_POST**, **\$_COOKIE**, **\$_SESSION**, **\$_FILES**, **\$_SERVER**, and **\$GLOBALS**.
- ▶ Through PHP, files can be read from and written to through creating file handle objects.
- ▶ The **mail** function can send emails.
- ▶ Much of the communication between JS and PHP is done through JSON objects.
- ▶ PHP is object oriented and allows for inheritance and the standard access modifiers.

Exercises I

1. What does PHP stand for?
2. How can global variables be accessed from within PHP functions?
3. If **x** is a PHP array, write a line of code to turn it into a string separated by "—"s.
4. Give an example of function overriding in the context of polymorphism for PHP.
5. Write an HTML form and PHP script that allows the user to enter a number, N. When the user submits the form, the page they see will produce a triangle pattern in asterisks starting at 1 in the top row, with 2 in the second row, all the way up to N in the Nth row. For example, if they enter 4 then the page should be displayed as:

*

**

Exercises II

6. Determine the truth values of:

```
true == 1  
true === 1  
false != 0  
false != 'false'  
true == []
```

7. What is the value of **\$f[2](3)**? Explain.

```
function foo($c){  
    $ret = [];  
    for($i = 1; $i <= $c; ++$i){  
        $ret[] = function($x) use ($i) { return $i; };  
    }  
    return $ret;  
}  
  
$f = foo(4);
```

Exercises III

8. Determine the values of **\$a**, **\$b**, **\$c**, and **\$arr** after the code below runs. Explain.

```
class X{  
    var $i;  
    public function __construct($j) { $this->i = $j;  
}
```

```
function bar($z){  
    ++$z[0];  
    ++$z[1];  
    ++$z[2]->i;  
    $z[3] = new X(40);  
    ++$z[3]->i;  
}
```

```
$a = 5;
```

```
$b = 7;
```

Exercises IV

```
$c = new X(4);
```

```
$d = new X(5);
```

```
$arr = [$a, &$amp;b, $c, $d];
```

```
bar($arr);
```


Exercises V

9. Write a PHP page that asks a user to solve 10 math addition problems, obtains their email address, and sends them the results. For the 10 problems, one at a time, it should ask the user to solve an addition problem displayed in a paragraph equipped with a text input and submit button, e.g. "[X] + [Y] = [input] [submit]" where [X] and [Y] are random integers from 0 to 100 inclusive, [input] represents an input box for the user, and [submit] is a button. After the 10 questions have been asked, it should ask them for their email address through a prompt. After they enter their email address, their score should be sent to them with subject "Math Score" and body "Your score was [N]/10" where [N] is the number of problems they solved correctly.

Exercises VI

10. Write a webpage, which may call upon a PHP script, that every time a user visits the page, their city is added to a local file. Every 10 seconds, the 5 most recent cities should be displayed, each in their own separate paragraph on the webpage.
11. Write a webpage where the user is given two buttons to press: “Valid” and “Invalid.” They are also given a “Redirect” button they can press.
When “Valid” is pressed, a cookie is added that will keep them in a “valid state” for 1 hour or until they press “Invalid”.
When “Invalid” is pressed, the cookie potentially storing a valid state is removed. By default if nothing has been pressed, the cookie should be absent.
When “Redirect” is pressed, they are taken to a page **validate.php** that either says “Valid” if they are in a valid state or “Invalid” if they are in an invalid state. *This message must be rendered in PHP, not JS!*

Exercises VII

12. Write a webpage where a user can upload *multiple image files*. The page should have a series of radio buttons where they can select from **png**, **jpg**, **eps**, and **pdf** as a common conversion type for all the files they selected. When they press a button saying “convert”, all of their images should be converted to the common type and downloaded. Use sessions to increase security so that the files can only be downloaded by the same user who uploads the original files.
13. Write a landing page **coords.html** with a canvas that is 200 pixels wide and tall with a thick black border. Then when a visitor clicks within the bounds of the canvas, they are taken to **coords.php** through a POST request where they see the x- and y-positions of their mouse click displayed on the page in a paragraph along with the x- and y-coordinates of all previous visitors' clicks. For example, if they clicked in the middle, they may see something like:
(100,100)
(53, 196)

Exercises VIII

on the **coords.php** page with the top result being where they clicked and the second result coming from a previous use of **coords.html**.