Session 4: Server-side programming

PHP Introduction

`PHP` is a widely-used open source general-purpose scripting language that is especially suited for web development and can be embedded into HTML. Nice, but what does that mean? An example:

> Example #1 An introductory example

```php
<?php
    echo "Hello, I'm a PHP script!";
?>
```

What distinguishes `PHP` from something like client-side `JavaScript` is that the code is executed on the server, generating `HTML` which is then sent to the client. The client would receive the results of running that script, but would not know what the underlying code was. You can even configure your web server to process all your `HTML` files with `PHP`, and then there's really no way that users can tell what you have up your sleeve. The best things in using `PHP` are that it is extremely simple for a newcomer, but offers many advanced features for a professional programmer. Don't be afraid reading the long list of PHP's features. You can jump in, in a short time, and start writing simple scripts in a few hours.

What can PHP do?

PHP is mainly focused on server-side scripting, so you can do anything any other CGI program can do, such as collect form data, generate dynamic page content, or send and receive cookies. But PHP can do much more.

There are three main areas where PHP scripts are used.

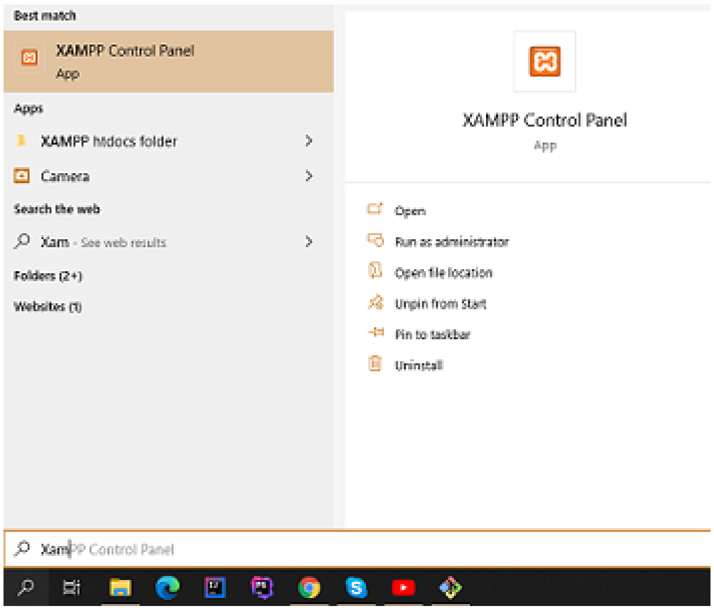
- Server-side scripting. This is the most traditional and main target field for PHP. You need three things to make this work: the PHP parser (CGI or server module), a web server and a web browser. You need to run the web server, with a connected PHP installation. You can access the PHP program output with a web browser, viewing the PHP page through the server. All these can run on your home machine if you are just experimenting with PHP programming.
- Command line scripting. You can make a PHP script to run it without any server or browser. You only need the PHP parser to use it this way. This type of usage is ideal for scripts regularly executed using cron (on *nix or Linux) or Task Scheduler (on Windows). These scripts can also be used for simple text processing tasks.
- Writing desktop applications. PHP is probably not the very best language to create a desktop application with a graphical user interface, but if you know PHP very well, and would like to use some advanced PHP features in your client-side applications you can also use PHP-GTK to write such programs. You also have the ability to write cross-platform applications this way. PHP-GTK is an extension to PHP, not available in the main distribution
- With PHP, you are not limited to output HTML. PHP's abilities includes outputting images, PDF files and even Flash movies (using libswf and Ming) generated on the fly. You can also output easily any text, such as XHTML and any other XML file. PHP can autogenerate these files, and save them in the file system, instead of printing it out, forming a server-side cache for your dynamic content.
- One of the strongest and most significant features in PHP is its support for a wide range of databases. Writing a database-enabled web page is incredibly simple using one of the database specific extensions (e.g., for mysql), or using an abstraction layer like PDO, or connect to any database supporting the Open Database Connection standard via the ODBC extension. Other databases may utilize cURL or sockets, like CouchDB.
- PHP also has support for talking to other services using protocols such as LDAP, IMAP, SNMP, NNTP, POP3, HTTP, COM (on Windows) and countless others. You can also open raw network sockets and interact using any other protocol. PHP has support for the WDDX complex data exchange between virtually all Web programming languages. Talking about interconnection, PHP has support for instantiation of Java objects and using them transparently as PHP objects.

Basic Syntax

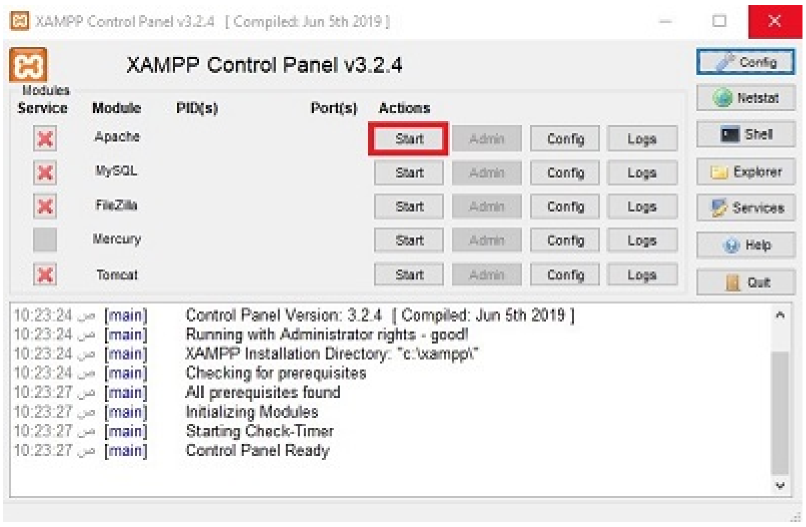
- Create a file named `hello.php` and put it in your web server's root directory (DOCUMENT_ROOT) in our case in `c:\xampp\htdocs\se2` then write the following content:

```
<html lang="en">
    <head>
        <title>PHP Test</title>
    </head>
    <body>
        <?php echo '<p>Hello World</p>'; ?>
    </body>
</html>
```

- Run Xampp control panel like this:

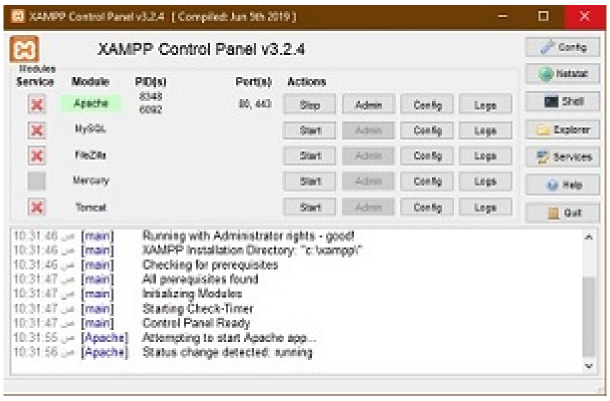


- Press on `Start`



```
10:23:24 ص  [main]   Control Panel Version: 3.2.4  [ Compiled: Jun 5th 2019 ]
10:23:24 ص  [main]   Running with Administrator rights - good!
10:23:24 ص  [main]   XAMPP Installation Directory: "c:\xampp\"
10:23:24 ص  [main]   Checking for prerequisites
10:23:27 ص  [main]   All prerequisites found
10:23:27 ص  [main]   Initializing Modules
10:23:27 ص  [main]   Starting Check-Timer
10:23:27 ص  [main]   Control Panel Ready
```

- You will see a green background on `Apache` , that means the apache server has started



```
10:31:46 ص [main]     Running with Administrator rights - good!
10:31:46 ص [main]     XAMPP Installation Directory: "c:\xampp\"
10:31:46 ص [main]     Checking for prerequisites
10:31:47 ص [main]     All prerequisites found
10:31:47 ص [main]     Initializing Modules
10:31:47 ص [main]     Starting Check-Timer
10:31:47 ص [main]     Control Panel Ready
10:31:55 ص [Apache]   Attempting to start Apache app...
10:31:56 ص [Apache]   Status change detected: running
```

- Use your browser to access the file with your web server's URL, ending with the /hello.php file reference. When developing locally this URL will be something like http://localhost/se2/hello.php or http://127.0.0.1/se2/hello.php but this depends on the web server's configuration. If everything is configured correctly, this file will be parsed by PHP and the following output will be sent to your browser:

```
<html lang="en">
    <head>
        <title>PHP Test</title>
    </head>
    <body>
        <p>Hello World</p>
    </body>
</html>
```

- This program is extremely simple, and you really did not need to use PHP to create a page like this. All it does is display: Hello World using the PHP `echo` statement. Note that the file does not need to be executable or special in any way. The server finds out that this file needs to be interpreted by PHP because you used the ".php" extension, which the server is configured to pass on to PHP. Think of this as a normal HTML file which happens to have a set of special tags available to you that do a lot of interesting things.
- When PHP parses a file, it looks for opening and closing tags, which are which tell PHP to start and stop interpreting the code between them. Parsing in this manner allows PHP to be embedded in all sorts of different documents, as everything outside a pair of opening and closing tags is ignored by the PHP parser. PHP includes a short echo tag <?= which is a short-hand to the more verbose <?php echo.
- Example #1 PHP Opening and Closing Tags

```
<?php
    echo 'if you want to serve PHP code in XHTML or XML documents,use these tags';
?>
```

- You can use the short echo tag to

```
    <?= 'print this string' ?>.
```

- PHP supports `C` , `C++` and `Unix` shell-style (Perl style) **comments**.For example:

```
<?php
    echo 'This is a test'; // This is a one-line c++ style comment
    /* This is a multi line comment
    yet another line of comment */
    echo 'This is yet another test';
    echo 'One Final Test'; # This is a one-line shell-style comment
?>
```

Types

- PHP supports ten primitive types.
- Four scalar types:
  - bool
  - int
  - float (floating-point number, aka double)
  - string
- Four compound types:
  - array
  - object
  - ~~callable~~
  - ~~iterable~~
- ~~Two special types:~~
  - resource
  - NULL

NOTE

> The type of variable is not usually set by the programmer; rather, it is decided at runtime by PHP depending on the context in which that variable is used. To check the type and value of an expression, use the var_dump() function. To get a human-readable representation of a type for debugging, use the gettype() function. To check for a certain type, do not use gettype(), but rather the is_type functions.

- Some examples:

```
<?php
    $a_bool = TRUE;   // a boolean
    $a_str  = "foo";  // a string
    $a_str2 = 'foo';  // a string
    $an_int = 12;     // an integer

    echo gettype($a_bool); // prints out:  boolean
    echo gettype($a_str);  // prints out:  string

    // If this is an integer, increment it by four
    if (is_int($an_int))
    {
        $an_int += 4;
    }

    // If $a_bool is a string, print it out
    // (does not print out anything)
    if (is_string($a_bool))
    {
        echo "String: $a_bool";
    }
?>
```

Variables

- Variables in PHP are represented by a dollar sign `$` followed by the name of the variable. The variable name is ***case-sensitive***. Variable names follow the same rules as other labels in PHP. A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: `^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$`
- Example:

```
<?php
    $var = 'Ali';
    $Var = 'Ibrahim';
    echo "$var, $Var";      // outputs "Ali, Ibrahim"

    $4site = 'not yet';     // invalid; starts with a number
    $_4site = 'not yet';    // valid; starts with an underscore
    $täyte = 'mansikka';    // valid; 'ä' is (Extended) ASCII 228.
?>
```

- ~~By default, variables are always assigned by value. That is to say, when you assign an expression to a variable, the entire value of the original expression is copied into the destination variable. This means, for instance, that after assigning one variable's value to another, changing one of those variables will have no effect on the other~~
- ~~PHP also offers another way to assign values to variables: assign by reference. This means that the new variable simply references (in other words, "becomes an alias for" or "points to") the original variable. Changes to the new variable affect the original, and vice versa.~~
- ~~To assign by reference, simply prepend an ampersand (&) to the beginning of the variable which is being assigned (the source variable). For instance, the following code snippet outputs 'My name is Bob' twice:~~

```
~~<?php~~
    ~~$foo = 'Bob';~~              ~~// Assign the value 'Bob' to $foo~~
    ~~$bar = &$foo;~~              ~~// Reference $foo via $bar.~~
    ~~$bar = "My name is $bar";~~  ~~// Alter $bar...~~
    ~~echo $bar;~~
    ~~echo $foo;~~                 ~~// $foo is altered too.~~
~~?>~~
```

- ~~**Note:** One important thing to note is that only named variables may be assigned by reference.~~

```php
~~<?php~~
    ~~$foo = 25;~~
    ~~$bar = &$foo;      // This is a valid assignment.~~
    ~~$bar = &(24 * 7);  // Invalid; references an unnamed expression.~~

    ~~function test()~~
    ~~{~~
        ~~return 25;~~
    ~~}~~

    ~~$bar = &test();    // Invalid.~~
~~?>~~
```

- ~~It is not necessary to initialize variables in PHP however it is a very **good practice**. Uninitialized variables have a `default` value of their type depending on the context in which they are used - booleans default to false, integers and floats default to zero, strings (e.g. used in echo) are set as an empty string and arrays become to an empty array. Example #1 Default values of uninitialized variables~~

```php
~~<?php~~
    ~~// Unset AND unreferenced (no use context) variable; outputs NULL~~
    ~~var_dump($unset_var);~~

    ~~// Boolean usage; outputs 'false' (See ternary operators for more on this syntax)~~
    ~~echo($unset_bool ? "true\n" : "false\n");~~

    ~~// String usage; outputs 'string(3) "abc"'~~
    ~~$unset_str .= 'abc';~~
    ~~var_dump($unset_str);~~

    ~~// Integer usage; outputs 'int(25)'~~
    ~~$unset_int += 25; // 0 + 25 => 25~~
    ~~var_dump($unset_int);~~

    ~~// Float/double usage; outputs 'float(1.25)'~~
    ~~$unset_float += 1.25;~~

    ~~var_dump($unset_float);~~

    ~~// Array usage; outputs array(1) {  [3]=>  string(3) "def" }~~
    ~~$unset_arr[3] = "def"; // array() + array(3 => "def") => array(3 => "def")~~
    ~~var_dump($unset_arr);~~

    ~~// Object usage; creates new stdClass object (see https://www.php.net/manual/en/reserved.class~~
     ~~// Outputs: object(stdClass)#1 (1) {  ["foo"]=>  string(3) "bar" }~~
    ~~$unset_obj->foo = 'bar';~~
    ~~var_dump($unset_obj);~~
~~?>~~
```

Variable scope

- The scope of a variable is the context within which it is defined. For the most part all PHP variables only have a single scope. This single scope spans included and required files as well. For example:

```php
<?php
    $a = 1;
    include 'b.inc';
?>
```

- Here the `$a` variable will be available within the included b.inc script. However, within user-defined functions a local function scope is introduced. Any variable used inside a function is by default limited to the local function scope. For example:

```php
<?php
    $a = 1; /* global scope */

    function test()
    {
        echo $a; /* reference to local scope variable */
    }

    test();
?>
```

- This script will not produce any output because the echo statement refers to a local version of the $a variable, and it has not been assigned a value within this scope. You may notice that this is a little different from the C language in that global variables in C are automatically available to functions unless specifically overridden by a local definition. This can cause some problems in that people may inadvertently change a global variable. In PHP global variables must be declared global inside a function if they are going to be used in that function.
- **The `global` keyword**
- Example #1 Using global:

```php
<?php
    $a = 1;
    $b = 2;

    function Sum()
    {
        global $a, $b;

        $b = $a + $b;
    }

    Sum();
    echo $b;
?>
```

- The above script will output 3. By declaring `$a` and `$b` global within the function, all references to either variable will refer to the global version. There is no limit to the number of global variables that can be manipulated by a function.
    - A second way to access variables from the global scope is to use the special PHP-defined `$GLOBALS` array. The previous example can be rewritten as:

- Example #2 Using ***$GLOBALS*** instead of global:

```php
<?php
    $a = 1;
    $b = 2;

    function Sum()
    {
        $GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
    }

    Sum();
    echo $b;
?>
```

- The `$GLOBALS` array is an associative array with the name of the global variable being the key and the contents of that variable being the value of the array element. Notice how `$GLOBALS` exists in any scope, this is because `$GLOBALS` is a superglobal.

```php
<?php
    function test_superglobal()
    {
        echo $_POST['name'];
    }
?>
```

- ~~**Using** `static` **variables**~~
  ~~Another important feature of variable scoping is the static variable. A static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope. Consider the following example:~~

```php
~~<?php~~
   ~~//Example #4: demonstrating need for static variables~~
   ~~function test()~~
   {
       ~~$a = 0~~;
       ~~echo $a~~;
        ~~$a++~~;
   }
~~?>~~
```

- ~~This function is quite useless since every time it is called it sets `$a` to 0 and prints 0. The `$a++` which increments the variable serves no purpose since as soon as the function exits the `$a` variable disappears. To make a useful counting function which will not lose track of the current count, the `$a` variable is declared static.~~

```php
~~<?php~~
    ~~//Example #5: Example use of static variables~~
    ~~function test()~~
    {
      ~~static $a = 0~~;
```

```
        echo $a;
         $a++;
     }
?>
```

- ~~Static variables also provide one way to deal with recursive functions. A recursive function is one which calls itself. Care must be taken when writing a recursive function because it is possible to make it recurse indefinitely. You must make sure you have an adequate way of terminating the recursion. The following simple function recursively counts to 10, using the static variable $count to know when to stop:~~

```
<?php
    //Example #6: Static variables with recursive functions
    function test()
    {
        static $count = 0;

        $count++;
        echo $count;
        if ($count < 10)
        {
            test();
        }
        $count--;
    }
?>
```

- ~~Static variables can be assigned values which are the result of constant expressions, but dynamic expressions, such as function calls, will cause a parse error.~~

```
<?php
//Example #7: Declaring static variables
    function foo()
    {
        static $int = 0;          // correct
         static $int = 1+2;        // correct
        static $int = sqrt(121);  // wrong  (as it is a function)
         $int++;
        echo $int;
    }
?>
```

> ~~**Note** Static declarations are resolved in compile-time.~~

- ~~You can read more about References with global and static variables ,Variable variables and Variables From External Sources.~~ ~~`Reading is optional`~~

Constant

- A constant is an identifier (name) for a simple value. As the name suggests, that value cannot change during the execution of the script (except for magic constants, which aren't actually constants). Constants are case-sensitive. By convention, constant identifiers are always uppercase.

> ***Note:*** Prior to PHP 8.0.0, constants defined using to define() function may be case-insensitive.

- The name of a constant follows the same rules as any label in PHP. A valid constant name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thusly: `^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$` .

- **Example Valid and invalid constant names**

```
<?php
    // Valid constant names
    define("FOO",     "something");
    define("FOO2",    "something else");
    define("FOO_BAR", "something more");

     // Invalid constant names
    define("2FOO",    "something");

    // This is valid, but should be avoided:
    // PHP may one day provide a magical constant
    // that will break your script
    define("__FOO__", "something");
 ?>
```

Functions

User-defined functions

- A function may be defined using syntax such as the following:
- **Example #1 Pseudo code to demonstrate function uses**

```
<?php
    function foo($arg_1, $arg_2, /* ..., */ $arg_n)
    {
        echo "Example function.\n";
        return $retval;
    }
?>
```

- Any valid PHP code may appear inside a function, even other functions and class definitions.
- Function names follow the same rules as other labels in PHP. A valid function name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: `^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$` .
- Functions need not be defined before they are referenced, ***except*** when a function is conditionally defined as shown in the two examples below.

- When a function is defined in a conditional manner such as the two examples shown. Its definition must be processed prior to being called.
- ~~**Example #2 Conditional functions**~~

```
~~<?php~~

    ~~$make_foo = true;~~

    ~~/* We can't call foo() from here since it doesn't exist yet,but we can call bar() */~~

     ~~bar();~~

    ~~if ($makefoo)~~
    {
        ~~function foo()~~
        {
            ~~echo "I don't exist until program execution reaches me.\n";~~
        }
    }

    ~~/* Now we can safely call foo() since $make_foo evaluated to true */~~

    ~~if ($makefoo) foo();~~

    ~~function bar()~~
    {
        ~~echo "I exist immediately upon program start.\n";~~
    }
~~?>~~
```

- ~~**Example #3 Functions within functions**~~

```
~~<?php~~
    ~~function foo()~~
    {
        ~~function bar()~~
        {
             ~~echo "I don't exist until foo() is called.\n";~~
        }
    }

    ~~/* We can't call bar() yet since it doesn't exist. */~~

    ~~foo();~~

    ~~/* Now we can call bar(),foo()'s processing has made it accessible. */~~

    ~~bar();~~
~~?>~~
```

- ~~All functions and classes in PHP have the global scope - they can be called outside a function even if they were defined inside and vice versa.~~
- ~~PHP does not support function overloading, nor is it possible to undefine or redefine previously-declared functions.~~

> ~~**Note:** Function names are case-insensitive for the ASCII characters A to Z, though it is usually good form to call functions as they appear in their declaration.~~

- ~~It is possible to call recursive functions in PHP.~~
- ~~**Example #4 Recursive functions**~~

```
~~<?php~~
   ~~function recursion($a)~~
   {
       ~~if ($a < 20)~~
       {
           ~~echo "$a\n";~~
           ~~recursion($a + 1);~~
       }
   }
~~?>~~
```

> ~~**Note:** Recursive function/method calls with over 100-200 recursion levels can smash the stack and cause a termination of the current script. Especially, infinite recursion is considered a programming error.~~

Function arguments

- PHP supports passing arguments by value (the default), passing by reference, and default argument values. Variable-length argument lists and Named Arguments are also supported
- **Example #1 Passing arrays to functions**

```
<?php
    function takes_array($input)
    {
        echo "$input[0] + $input[1] = ", $input[0]+$input[1];
    }
?>
```

- The list of function arguments may include a trailing comma, which will be ignored. That is particularly useful in cases where the list of arguments is long or contains long variable names, making it convenient to list arguments vertically.
- **Example #2 Function Argument List with trailing Comma**

```
<?php
    function takes_many_args($first_arg,$second_arg,$a_very_long_argument_name,$arg_with_default =
    {
     // ...
    }
?>
```

- ~~Passing mandatory arguments after optional arguments is deprecated. This can generally be resolved by dropping the default value. One exception to this rule are arguments of the form Type~~

~~$param = null, where the null default makes the type implicitly nullable. This usage remains allowed, though it is recommended to use an explicit nullable type instead.~~

- ~~**Example #3 Passing optional arguments after mandatory arguments**~~

```
<?php
    function foo($a = [], $b) {} // Before
    function foo($a, $b) {}      // After
    function bar(A $a = null, $b) {} // Still allowed
    function bar(?A $a, $b) {}       // Recommended
?>
```

- ~~Read more about passing argument~~ ~~`Recommended`~~

Returning value

- Values are returned by using the optional return statement. Any type may be returned, including arrays and objects. This causes the function to end its execution immediately and pass control back to the line from which it was called. See return for more information.

> ***Note*** If the return is omitted the value null will be returned.

- **Use of return:**
  - **Example #1:**

```
<?php
    function square($num)
    {
        return $num * $num;
    }
    echo square(4);   // outputs '16'.
?>
```

- A function can not return multiple values, but similar results can be obtained by returning an array.
- **Example #2 Returning an array to get multiple values**

```
<?php
    function small_numbers()
    {
        return [0, 1, 2];
    }
    // Array destructuring will collect each member of the array individually
    [$zero, $one, $two] = small_numbers();

    // Prior to 7.1.0, the only equivalent alternative is using list() construct
    list($zero, $one, $two) = small_numbers();
?>
```

~~Variable functions~~

- ~~PHP supports the concept of variable functions. This means that if a variable name has parentheses appended to it, PHP will look for a function with the same name as whatever the variable evaluates to, and will attempt to execute it. Among other things, this can be used to implement callbacks, function tables, and so forth.~~
- ~~Variable functions won't work with language constructs such as echo, print, unset(), isset(), empty(), include, require and the like. Utilize wrapper functions to make use of these constructs as variable functions~~
- **~~Example #1 Variable function example~~**

```
<?php
    function foo()
    {
        echo "In foo()<br />\n";
    }

    function bar($arg = '')
    {
        echo "In bar(); argument was '$arg'.<br />\n";
    }

    // This is a wrapper function around echo
    function echo_it($str)
    {
        echo $str;
    }

    $func = 'foo';
    $func();        // This calls foo()

    $func = 'bar';
    $func('test');  // These calls bar()

    $func = 'echo_it';
    $func('test');  // These calls echo_it()
?>
```

- **~~Example #2 Variable method example~~**

```
<?php
     class Foo
     {
        function Variable()
        {
            $name = 'Bar';
            $this->$name(); // This calls the Bar() method
        }

        function Bar()
        {
            echo "This is Bar";
        }
     }

    $foo = new Foo();
    $func_name = "Variable";
    $foo->$func_name();  // This calls $foo->Variable()
?>
```

- ~~**Example #3 Variable method example with static properties**~~

```
<?php
    class Foo
    {
        static $variable = 'static property';
        static function Variable()
        {
            echo 'Method Variable called';
        }
    }

    echo Foo::$variable; // This prints 'static property'. It does need a $variable in this scope.
    $variable = "Variable";
    Foo::$variable();  // This calls $foo->Variable() reading $variable in this scope.
?>
```

- ~~**Example #4 Complex callables**~~

```
<?php
    class Foo
        {
            static function bar()
            {
                echo "bar\n";
            }
            function baz()
            {
                echo "baz\n";
            }
        }

    $func = array("Foo", "bar");
    $func(); // prints "bar"
    $func = array(new Foo, "baz");
    $func(); // prints "baz"
    $func = "Foo::bar";
    $func(); // prints "bar"
?>
```

Object-Oriented Concepts

Before we go in detail, lets define important terms related to Object-Oriented Programming:

- **Class**: This is a programmer-defined data type, which includes local functions as well as local data. You can think of a class as a template for making many instances of the same kind (or class) of object.
- **Object**: An individual instance of the data structure defined by a class. You define a class once and then make many objects that belong to it. Objects are also known as instance.

- **Member Variable**: These are the variables defined inside a class. This data will be invisible to the outside of the class and can be accessed via member functions. These variables are called attribute of the object once an object is created
- **Member function**: These are the function defined inside a class and are used to access object data.
- **Inheritance**: When a class is defined by inheriting existing function of a parent class then it is called inheritance. Here child class will inherit all or few member functions and variables of a parent class.
- **Parent class**: A class that is inherited from by another class. This is also called a base class or super class.
- **Child Class**: A class that inherits from another class. This is also called a subclass or derived class.
- **Polymorphism**: This is an object-oriented concept where same function can be used for different purposes. For example function name will remain same, but it takes different number of arguments and can do different task.
- **Overloading**: a type of polymorphism in which some or all operators have different implementations depending on the types of their arguments. Similarly, functions can also be overloaded with different implementation.
- **Data Abstraction**: Any representation of data in which the implementation details are hidden (abstracted).
- **Encapsulation**: refers to a concept where we encapsulate all the data and member functions together to form an object.
- **Constructor**: refers to a special type of function which will be called automatically whenever there is an object formation from a class.
- **Destructor**: refers to a special type of function which will be called automatically whenever an object is deleted or goes out of scope.

Defining PHP Classes

- Basic class definitions begin with the keyword class, followed by a class name, followed by a pair of curly braces which enclose the definitions of the properties and methods belonging to the class.
- The class name can be any valid label, provided it is not a PHP reserved word. A valid class name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: `^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$` .
- A class may contain its own constants, variables (called `properties`), and functions (called `methods`).
- **Example #1 Simple Class definition**

```
<?php
   class Books {
      /* Member variables */
      var $price;
      var $title;

      /* Member functions */
      function setPrice($par)
      {
         $this->price = $par;
      }

      function getPrice()
      {
         echo $this->price ."<br/>";
      }
      function setTitle($par)
      {
         $this->title = $par;
      }

      function getTitle()
      {
         echo $this->title ." <br/>";
      }
   }
?>
```

- The pseudo-variable `$this` is available when a method is called from within an object context. `$this` is the value of the calling object.
- To create an instance of a class, the new keyword must be used. An object will always be created unless the object has a constructor defined that throws an exception on error. Classes should be defined before instantiation (and in some cases this is a requirement).
- If a string containing the name of a class is used with new, a new instance of that class will be created. If the class is in a namespace, its fully qualified name must be used when doing this.
- Once you defined your class, then you can create as many objects as you like of that class type. Following is an example of how to create object using new operator.
- **Example #2 Creating an instance**

```
<?php
    $physics = new Books();
    $maths = new Books();
    $chemistry = new Books();
?>
```

- Here we have created three objects and these objects are independent of each other, and they will have their existence separately. Next we will see how to access member function and process member variables.
- After creating your objects, you will be able to call member functions related to that object. One member function will be able to process member variable of related object only.

Following example shows how to set title and prices for the three books by calling member functions.

- **Example #3 Calling Member Functions**

```
<?php
    $physics->setTitle( "Physics for High School" );
    $chemistry->setTitle( "Advanced Chemistry" );
    $maths->setTitle( "Algebra" );

    $physics->setPrice( 10 );
    $chemistry->setPrice( 15 );
    $maths->setPrice( 7 );
?>
```

- Now you call another member functions to get the values set by in above example

- **Example #4 Calling Member Functions**

```
<?php
    $physics->getTitle();
    $chemistry->getTitle();
    $maths->getTitle();
    $physics->getPrice();
    $chemistry->getPrice();
    $maths->getPrice();
?>
```

- The above example will output:

```
Physics for High School
Advanced Chemistry
Algebra
10
15
7
```

Constructor Functions

- Constructor Functions are special type of functions which are called automatically whenever an object is created. So we take full advantage of this behaviour, by initializing many things through constructor functions.
- PHP provides a special function called __construct() to define a constructor. You can pass as many as arguments you like into the constructor function.
- Following example will create one constructor for Books class, and it will initialize price and title for the book at the time of object creation.
- **Example #5 define constructor**

```
<?php
    function __construct( $par1, $par2 )
    {
        $this->title = $par1;
        $this->price = $par2;
    }
?>
```

- Now we don't need to call set function separately to set price and title. We can initialize these two member variables at the time of object creation only. Check following example below

```
<?php
    $physics = new Books( "Physics for High School", 10 );
    $maths = new Books ( "Advanced Chemistry", 15 );
    $chemistry = new Books ("Algebra", 7 );

    /* Get those set values */
    $physics->getTitle();
    $chemistry->getTitle();
    $maths->getTitle();
    $physics->getPrice();
    $chemistry->getPrice();
    $maths->getPrice();
?>
```

- This will produce the following result:

```
Physics for High School
Advanced Chemistry
Algebra
10
15
7
```

~~Inheritance~~

- ~~PHP class definitions can optionally inherit from a parent class definition by using the extends clause. The syntax is as follows~~

```
<?php
    class Child extends Parent
    {
        <definition body>
    }
?>
```

- ~~The effect of inheritance is that the child class (or subclass or derived class) has the following characteristics −~~
    - ~~Automatically has all the member variable declarations of the parent class.~~
    - ~~Automatically has all the same member functions as the parent, which (by default) will work the same way as those functions do in the parent.~~
- ~~Following example inherit Books class and adds more functionality based on the requirement~~

```
<?php
    class Novel extends Books
    {
        var $publisher;

        function setPublisher($par)
        {
            $this->publisher = $par;
        }

        function getPublisher()
        {
            echo $this->publisher. "<br />";
        }
    }
?>
```

Public, Private and Protected Members

- Unless you specify otherwise, properties and methods of a class are public. That is to say, they may be accessed in three possible situations:
    - From outside the class in which it is declared.
    - From within the class in which it is declared.
    - From within another class that implements the class in which it is declared
- By designating a member private, you limit its accessibility to the class in which it is declared. The private member cannot be referred to from classes that inherit the class in which it is declared and cannot be accessed from outside the class.
- A class member can be made private by using private keyword informant of the member.
- Till now, we have seen all members as public members. If you wish to limit the accessibility of the members of a class then you define class members as private or protected.

```
<?php
    class MyClass
    {
        private $car = "skoda";
        $driver = "SRK";

        function __construct($par)
        {
            // Statements here run every time
            // an instance of the class
            // is created.
        }

        function myPublicFunction()
        {
            return("I'm visible!");
        }

        private function myPrivateFunction()
        {
            return("I'm  not visible outside!");
        }
    }
?>
```

- When `MyClass` class is inherited by another class using extends, `myPublicFunction()` will be visible, as will `$driver` . The extending class will not have any awareness of or access to myPrivateFunction and `$car` , because they are declared private
- A protected property or method is accessible in the class in which it is declared, as well as in classes that extend that class. Protected members are not available outside those two kinds of classes. A class member can be made protected by using protected keyword in front of the member.

```
    class MyClass
    {
        protected $car = "skoda";
        $driver = "SRK";
        function __construct($par)
        {
            // Statements here run every time
            // an instance of the class
            // is created.
        }

        function myPublicFunction()
        {
            return("I'm visible!");
        }

        protected function myPrivateFunction()
        {
            return("I'm  visible in child class!");
        }
    }
```

~~static and final keyword~~

- ~~Declaring class members or methods as static makes them accessible without needing an instantiation of the class. A member declared as static can not be accessed with an instantiated class object (though a static method can).~~
- ~~Try out following example~~

```
~~<?php~~
   ~~class Foo~~
   ~~{~~
      ~~public static $my_static = 'foo';~~

      ~~public function staticValue()~~
      ~~{~~
         ~~return self::$my_static;~~
      ~~}~~
   ~~}~~

   ~~print Foo::$my_static . "\n";~~
   ~~$foo = new Foo();~~

   ~~print $foo->staticValue() . "\n";~~
~~?>~~
```

- ~~From PHP 5 introduces the final keyword, which prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended.~~
- ~~Following example results in Fatal error: Cannot override final method BaseClass::moreTesting()~~

```
~~<?php~~
   ~~class BaseClass~~
   ~~{~~
      ~~public function test()~~
      ~~{~~
         ~~echo "BaseClass::test() called<br>";~~
      ~~}~~
```

```
        final public function moreTesting()
        {
            echo "BaseClass::moreTesting() called<br>";
        }
    }

    class ChildClass extends BaseClass
    {
        public function moreTesting()
        {
            echo "ChildClass::moreTesting() called<br>";
        }
    }
?>
```

More

You can read about Expressions, Operators, Error handling and Control Structure ...etc.