WORDPRESS

CS-16: Content Management System using WordPress

B.C.A. Semester - 3

Topic-1

OOP (Object Oriented Programming)

- Concept of OOP
  - Class
  - Property
  - Visibility
  - Constructor
  - Destructor
  - Inheritance
  - Scope Resolution Operator (::)
  - Autoloading Classes
  - Class Constants
- MySQL Database handling with OOPs (insert, update, select, delete)

Mr. Gaurav.k.sardhara

9067351366

We can imagine our universe made of different objects like sun, earth, moon etc. Similarly we can imagine our car made of different objects like wheel, steering, gear etc. Same way there is object oriented programming concepts which assume everything as an object and implement a software using different objects.

Core concept

While classes and the entire concept of Object Oriented Programming (OOP) is the basis of lots of modern programming languages, PHP was built on the principles of functions instead.

Basic support for classes was first introduced in version 4 of PHP but then re-written for version 5, for a more complete OOP support.

Today, PHP is definitely usable for working with classes, and while the PHP library still mainly consists of functions, classes are now being added for various purposes. However, the main purpose is of course to write and use your own classes.

Classes can be considered as a collection of methods, variables and constants. They often reflect a real-world thing, like a Car class or a Fruit class. You declare a class only once, but you can instantiate as many versions of it as can be contained in memory. An instance of a class is usually referred to as an object.

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.

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.

A class may contain its own constants, variables (called "properties"), and functions (called "methods").

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

```
<?php
        class User  {       }
?>
```

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

```
        $user = new User();
```

But since the class can't do anything yet, the $user object is just as useless. Let's remedy (prepare) that by adding a couple of class variables and a method: @ 01_oop_basics.php

```
class User
{       public $name;
        public $age;
        public function Describe()
        {       return $this->name." is ".$this->age." years old";      }
}
```

Okay, there are a couple of new concepts here. First of all, we declare two class variables, a name and an age. The variable name is prefixed by the access modifier "public", which basically means that the variable can be accessed from outside the class.

Next, we define the Describe() function. As you can see, it looks just like a regular function declaration, but with a couple of exceptions. It has the public keyword in front of it, to specify the access modifier. Inside the function, we use the "$this" variable, to access the variables of the class itself. $this is a special variable

in PHP, which is available within class functions and always refers to the object from which it is used.

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

```
$user = new User();
$user->name = "John Doe";
$user->age = 42;
echo $user->Describe();
```

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

```
John Doe is 42 years old
```

Congratulations, you have just defined and used your first class, but there is much more to classes than this

Object Oriented Concepts

Before we go in detail, let's 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 make take different number of arguments and can do different task.
- Overloading: a type of polymorphism in which some or all of 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

The general form for defining a new class in PHP is as follows −

```
<?php
class phpClass
{
        var $var1;
        var $var2 = "constant string";
        function myfunc ($arg1, $arg2) {      [..]      }
        [..]
}
?>
```

Here is the description of each line −

- The special form class, followed by the name of the class that you want to define.
- A set of braces enclosing any number of variable declarations and function definitions.
- Variable declarations start with the special form var, which is followed by a conventional $ variable name; they may also have an initial assignment to a constant value.
- Function definitions look much like standalone PHP functions but are local to the class and will be used to set and access object data.

Example: Here is an example which defines a class of Books type. @ 02_class_get_set.php

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

        /* Member functions */
        function setTitle($par) {       $this->title = $par;      }
        function getTitle()
        {       echo "<b>Book Title : </b>".$this->title."<br>";        }

        function setPrice($par) {       $this->price = $par;      }
        function getPrice()
        {       echo "<b>Price: Rs. </b>".$this->price."<br><br>";      }
}
```

The variable $this is a special variable and it refers to the same object ie. Itself.

Creating Objects using “new” keyword

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.

```
$physics = new Books;
$chemistry = new Books;
$maths = 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.

Calling Member Functions

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.

```
$physics->setTitle ("Physics for High School");
$physics->setPrice (1000);
$chemistry->setTitle ("Advanced Chemistry");
$chemistry->setPrice (1500);
$maths->setTitle ("Algebra");
$maths->setPrice (700);
```

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

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

This will produce the following result –

```
Book Title: Physics for High School
Price: Rs. 1000
Book Title: Advanced Chemistry
Price: Rs. 1500
Book Title: Algebra
Price: Rs. 700
```

Properties

Class member variables are called "properties". You may also see them referred to using other terms such as "attributes" or "fields", but for the purposes of this reference we will use "properties".

They are defined by using one of the keywords public, protected, or private, followed by a normal variable declaration. This declaration may include an initialization, but this initialization must be a constant value--that is, it must be able to be evaluated at compile time and must not depend on run-time information in order to be evaluated.

Within class methods non-static properties may be accessed by using "->" (Object Operator):

```
$this->property        // where property is the name of the property (member variable).
```

Static properties are accessed by using the "::" (Double Colon): self::$property

The pseudo-variable $this is available inside any class method when that method is called from within an object context. $this is a reference to the calling object (usually the object to which the method belongs, but possibly another object, if the method is called statically from the context of a secondary object).

Example: property declarations

```
class SimpleClass
{
    // valid as of PHP 5.6.0:
    public $var1 = 'hello ' . 'world';

    // valid as of PHP 5.3.0:
    public $var2 = <<<EOD
                    hello world
                    EOD;
    // valid as of PHP 5.6.0:
    public $var3 = 1+2;

    // invalid property declarations:
    public $var4 = self::myStaticMethod();
    public $var5 = $myVar;

    // valid property declarations:
    public $var6 = myConstant;
    public $var7 = array(true, false);

    // valid as of PHP 5.3.0:
    public $var8 = <<<'EOD'
                    hello world
                    EOD;
}
```

Visibility

Visibility is a big part of OOP. It allows you to control where your class members can be accessed from, for instance to prevent a certain variable to be modified from outside the class. The default visibility is public, which means that the class members can be accessed from anywhere. This means that declaring the visibility is optional, since it will just fall back to public if there is no access modifier. For backwards compatibility, the old way of declaring a class variable, where you would prefix the variable name with the "var" keyword (this is from PHP 4 and should not be used anymore) will also default to public visibility.

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

Public 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

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.

Private members

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 in front of the member.

@ 03_private_access.php

```
class MyClass
{
      private $car = "Hyundai";
      var $model;

      function_construct($par)
      {     $this->model = $par;     }

      function myPublicFunction()
      {     return ("I'm visible! <br> Car: ".$this->car." ".$this->model);    }

      private function myPrivateFunction()
      {  return ("I'm not visible outside! <br> Car: ".$this->car." ".$this->model);   }
}

$obj = new MyClass("Creta");
echo $obj->myPublicFunction()."<br>";

/* Fatal error: Call to private method MyClass::myPrivateFunction() */
echo $obj->myPrivateFunction();
```

When MyClass class is inherited by another class using extends, myPublicFunction() will be visible, as will $model. The extending class will not have any awareness of or access to myPrivateFunction and $car, because they are declared private.

Protected members

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 of those two kinds of classes. A class member can be made protected by using protected keyword in front of the member.

Here is different version of MyClass. @ 04_protected_access.php

```
class MyClass
{       protected $car = "Maruti";
        var $model;

        function_construct($par)
        {       $this->model = $par;     }

        function myPublicFunction()
        {       return ("I'm visible! <br> Car: ".$this->car." ".$this->model);    }

        protected function myPrivateFunction()
        {  return ("I'm not visible outside! <br> Car: ".$this->car." ".$this->model);   }
}
$obj = new MyClass("Baleno");
echo $obj->myPublicFunction()."<br>";

/* Fatal error: Call to protected method MyClass::myPrivateFunction() */
echo $obj->myPrivateFunction();
```

Magic Functions

PHP functions that start with a double underscore – a "_" – are called magic functions (and/or methods) in PHP. They are functions that are always defined inside classes, and are not stand-alone (outside of classes) functions. The magic functions available in PHP are: __construct(), __destruct(), _call(), __callStatic(), __get(), __set(), __isset(), __unset(), __sleep(), __wakeup(), __toString(), __invoke(),_set_state(), __clone(), and_autoload().

The definition of a magic function is provided by the programmer – meaning you, as the programmer, will actually write the definition. This is important to remember – PHP does not provide the definitions of the magic functions – the programmer must actually write the code that defines what the magic function will do. But, magic functions will never directly be called by the programmer – actually, PHP will call the function 'behind the scenes'. This is why they are called 'magic' functions – because they are never directly called, and they allow the programmer to do some pretty powerful things. Confused? An example will help make this clear.

Example of using the_construct() magic function in PHP

The most commonly used magic function is_construct(). This is because as of PHP version 5, the __construct method is basically the constructor for your class. If PHP 5 cannot find the_construct() function for a given class, then it will search for a function with the same name as the class name – this is the old way of writing constructors in PHP, where you would just define a function with the same name as the class.

Now, here is an example of a class with the_construct() magic function:

```
class Animal
{       public $height;   // height of animal
        public $weight;   // weight of animal
        public function __construct($height, $weight)
        {
                $this->height = $height;       //set the height instance variable
                $this->weight = $weight;       //set the weight instance variable
        }
}
```

In the code above, we have a simple_construct function defined that just sets the height and weight of an animal object. So let's say that we create an object of the Animal class with this code:

```
        Animal obj = new Animal (5, 150);
```

What happens when we run the code above? Well, a call to the__construct() function is made because that is the constructor in PHP 5. And the obj object will be an object of the Animal class with a height of 5 and a weight of 150. So, the_construct function is called behind the scenes. Magical, isn't it?

Constructor

A constructor and a destructor are special functions which are automatically called when an object is created and destroyed. The constructor is most useful of the two, especially because it allows you to send parameters along when creating a new object, which can then be used to initialize variables on the object. Here's an example of a class with a simple constructor: @ 05_constructor_default.php

```
class Animal
{
	public $name = "No-name animal";
	public function_construct()
	{	echo "I'm alive! <br>My name is ".$this->name;  }
}
```

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

```
	$animal = new Animal();
```

With just that line of code, the object will be created, the constructor called and the lines of code in it executed, which will cause our "I'm alive!" line to be outputted. However, as mentioned previously, a big advantage of the constructor is the ability to pass parameters which can be used to initialize member variables. Let's try doing just that: @ 06_constructor_param.php

```
class Animal
{
	public $name = "No-name animal";
	public function_construct($par)
	{	$this->name = $par;	}
}
$animal = new Animal("Tommy");
echo "My name is ".$animal->name;
```

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

Calling parent constructors

Instead of writing an entirely new constructor for the subclass, let's write it by calling the parent's constructor explicitly and then doing whatever is necessary in addition for instantiation of the subclass. Here's a simple example @ 07_constructor_classname.php

```
class Name
{	var $_firstName;
	var $_lastName;

	function Name($first_name, $last_name)
	{
		$this->_firstName = $first_name;
		$this->_lastName = $last_name;
	}

	function toString()
	{	return($this->_lastName .", " .$this->_firstName);	}
}
class NameSub1 extends Name
{
	var $_middleInitial;
```

```
        function NameSub1($first_name, $middle_initial, $last_name)
        {
                Name::Name($first_name, $last_name);
                $this->_middleInitial = $middle_initial;
        }
        function toString()
        {       return(Name::toString() . " " . $this->_middleInitial);      }
}
$obj = new NameSub1("Mehul", "D.", "Dhokia");
echo $obj->toString();
```

In this example, we have a parent class (Name), which has a two-argument constructor, and a subclass (NameSub1), which has a three-argument constructor. The constructor of NameSub1 functions by calling its parent constructor explicitly using the double-colon "::" syntax (passing two of its arguments along) and then setting an additional field. Similarly, NameSub1 defines its non-constructor toString() function in terms of the parent function that it overrides.

NOTE – A constructor can be defined with the same name as the name of a class. It is defined in above example.

The above example is also work with magic function with the_construct (), yes you heard it correct. Simply you just need to modify the Class name in constructor functions declaration to_construct (). Also while calling the constructor of parent class as Name::_construct (). It will work definitely.

Let's see with an example @ 08_constructor_construct().php

Destructor

A destructor is called when the object is destroyed. In some programming languages, you have to manually dispose of objects you created, but in PHP, it's handled by the Garbage Collector, which keeps an eye on your objects and automatically destroys them when they are no longer needed. Have a look at the following example, which is an extended version of our previous example: @ 09_destructor.php

```
class Animal
{       public $name = "No-name animal";
        public function __construct($name)
        {       echo "I'm alive!";
                $this->name = $name;     }
        public function_destruct()
        {       echo "I'm dead now :(";        }
}
$animal = new Animal("Bob");
echo "<br>Name of the animal: ".$animal->name."<br>";
```

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

Inheritance

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

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

child class for each specific animal you need. Another advantage to this approach is that you will every animal you have will come with the same basic functionality that you can always rely on.

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

```
class Animal
{
	public $name;
	public function Greet()
	{  return "Hello, I'm some sort of animal and my name is ".$this->name;  }
}
```

Notice that we declare the Greet() function again, because we need for it to do something else, but the $name class variable is not declared - we already have that defined on the Animal class, which is just fine. As you can see, even though $name is not declared on the Dog class, we can still use it in its Greet() function. Now, with both classes declared, it's time to test them out. The following code will do that for us:

```
$animal = new Animal();
echo $animal->Greet();
$animal = new Dog();
$animal->name = "Bob";
echo $animal->Greet();
```

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

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

Another concept

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

```
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.

@ 10_inheritance.php

```
class Novel extends Books
{	var $publisher;
	function setPublisher($par)
	{	$this->publisher = $par;	}

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

Now apart from inherited functions, class Novel keeps two additional member functions.

Function Overriding

Function definitions in child classes override definitions with the same name in parent classes. In a child class, we can modify the definition of a function inherited from parent class.

In the following example getTitle and getPrice functions are overridden to return some values.

@ 11_override.php

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

Scope resolution operator (::)

The Scope Resolution Operator or in simpler terms, the double colon, is a token that allows access to static, constant, and overridden properties or methods of a class.

When referencing these items from outside the class definition, use the name of the class.

Example-1: from outside the class definition

```
class MyClass
{	const CONST_VALUE = 'A constant value';	}

$classname = 'MyClass';
echo $classname::CONST_VALUE;		// As of PHP 5.3.0

echo MyClass::CONST_VALUE;
```

Three special keywords self, parent and static are used to access properties or methods from inside the class definition

Example-2: from inside the class definition

```
class OtherClass extends MyClass
{
	public static $my_static = 'static var';
	public static function doubleColon()
	{
		echo parent::CONST_VALUE . "\n";
		echo self::$my_static . "\n";
	}
}
$classname = 'OtherClass';
$classname::doubleColon();		// As of PHP 5.3.0

OtherClass::doubleColon();
```

When an extending class overrides the parent's definition of a method, PHP will not call the parent's method. It's up to the extended class on whether or not the parent's method is called. This also applies to Constructors and Destructors, Overloading, and Magic method definitions.

Example-3: Calling a parent's method

```
class MyClass
{	protected function myFunc()
	{	echo "MyClass::myFunc()\n";	}
}
```

```
class OtherClass extends MyClass
{     // Override parent's definition
      public function myFunc()
      {     // But still call the parent function
            parent::myFunc();
            echo "OtherClass::myFunc()\n";
      }
}
$class = new OtherClass();
$class->myFunc();
```

A class constant, class property (static), and class function (static) can all share the same name and be accessed using the double-colon.

```
class A
{
      public static $B = '1';          // Static class variable.
      const B = '2';                   // Class constant.
      public static function B()
      {     // Static class function.
            return '3';
      }
}
echo A::$B . A::B . A::B();            // Outputs: 123
```

Interfaces

Interfaces are defined to provide a common function names to the implementers. Different implementers can implement those interfaces according to their requirements. You can say, interfaces are skeletons which are implemented by developers.

As of PHP5, it is possible to define an interface, like this – @ 12_interface.php

```
interface Mail
{
      public function sendMail();
}
```

Then, if another class implemented that interface, like this –

```
class Report implements Mail
{
      // sendMail() Definition goes here
}
```

Abstract classes

Abstract classes are special because they can never be instantiated. Instead, you typically inherit a set of base functionality from them in a new class. For that reason, they are commonly used as the base classes in a larger class hierarchy.

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

A method can be marked as abstract as well. As soon as you mark a class function as abstract, you have to define the class as abstract as well - only abstract classes can hold abstract functions. Another consequence is that you don't have to (and can't) write any code for the function - it's a declaration only. You would do this to force anyone inheriting from your abstract class to implement this function and write the proper code for it. If you don't, PHP will throw an error. However, abstract classes can also contain non-abstract methods, which allows you to implement basic functionality in the abstract class.

Let's go on with an example. Here is the abstract class: @ 13_abstract.php

```
abstract class Animal
{	public $name;
	public $age;
	public function Describe()
	{	return $this->name." ".$this->age." years old";	}
	abstract public function Greet();
}
```

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

```
class Dog extends Animal
{
	public function Greet()
	{	return "Woof! Woof!";	}
	public function Describe()
	{	return parent::Describe()." and I'm a dog!<br>";	}
}
```

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

In this case, we use the parent keyword to reference the Animal class, and then we call Describe() function on it. We then add some extra text to the result, to clarify which type of animal we're dealing with. Now, let's try using this new class:

```
	$animal = new Dog();
	$animal->name = "Frodo";
	$animal->age = 7;
	echo $animal->Describe();
	echo $animal->Greet();
```

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

Static classes

Since a class can be instantiated more than once, it means that the values it holds, are unique to the instance/object and not the class itself. This also means that you can't use methods or variables on a class without instantiating it first, but there is an exception to this rule. Both variables and methods on a class can be declared as static (also referred to as "shared" in some programming languages), which means that they can be used without instantiating the class first. Since this means that a class variable can be accessed without a specific instance, it also means that there will only be one version of this variable. Another consequence is that a static method cannot access non-static variables and methods, since these require an instance of the class.

In a previous discussion, we wrote a User class. Let's expand it with some static functionality, to see what the fuzz (down) is all about: @ 14_static.php

```
<?php
class User
{	public $name;
	public $age;
```

```
        public static $minimumPasswordLength = 6;

        public function Describe()
        {       return $this->name." is ".$this->age." years old";      }

        public static function ValidatePassword($password)
        {       if(strlen($password) >= self::$minimumPasswordLength)
                        return true;
                else
                        return false;
        }
}
$password = "test";
if(User::ValidatePassword($password))
        echo "Password is valid!";
else
        echo "Password is NOT valid!";
?>
```

What we have done to the class, is adding a single static variable, the $minimumPasswordLength which we set to 6, and then we have added a static function to validate whether a given password is valid. I admit that the validation being performed here is very limited, but obviously it can be expanded. Now, couldn't we just do this as a regular variable and function on the class? Sure we could, but it simply makes more sense to do this statically, since we don't use information specific to one user - the functionality is general, so there's no need to have to instantiate the class to use it.

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

Autoloading Class

Why is the_autoload function used?

In PHP, the_autoload function is used to simplify the job of the programmer by including classes automatically without the programmer having to add a very large number of include statements. An example will help clarify. Suppose we have the following code:

```
include "class/class.Foo.php";
include "class/class.AB.php";
include "class/class.XZ.php";
include "class/class.YZ.php";

$foo = new Foo;
$ab = new AB;
$xz = new XZ;
$yz = new YZ;
```

Note in the code above that we have to include each of the 4 different class files separately – because we are creating an instance of each class, we absolutely must have each class file. Of course, we are assuming that developers are defining only one class per source file – which is good practice when writing object oriented programs, even though you are allowed to have multiple classes in one source file.

The_autoload function simplifies inclusion of class files in PHP

Imagine if we need to use 20 or even 30 different classes within this one file – writing out each include statement can become a huge pain. And this is exactly the problem that the PHP __autoload function solves – it allows PHP to load the classes for us automatically! So, instead of the code above, we can use the __autoload function as shown below:

```
function __autoload($class_name)
{
      require_once "./classes.".$class_name."-php";
}
$foo = new Foo;
$ab = new AB;
$xz = new XZ;
$yz = new YZ;
```

How does the_autoload function work?

Because the_autoload function is a magic function, it will not be called directly by you, the programmer. Instead, it is called behind the scenes by PHP – that's what makes it magical. But, when does the_autoload function actually get called? Well, in the code above, the_autoload function will be called 4 times, because PHP will not recognize the Foo, AB, XZ, and YZ classes so PHP will make a call to the_autoload function each time it does not recognize a class name.

Also, in the code above, we can see that the autoload function takes the class name as a parameter (the $class_name variable). PHP passes the $class_name variable behind the scenes to the_autoload function whenever it finds that it doesn't recognize the class name that is being used in a given statement. For instance, when PHP sees the "$foo = new Foo;" line, it does not recognize the Foo class because the Foo class was never included or "required" as part of the current file. So, PHP then (behind the scenes) passes the "Foo" class to the_autoload function, and if the class file is found by the autoload function then it is included by the "require_once" statement.

One last thing worth noticing in the code above is how we use the $class_name variable in the class.".$class_name."-php"; piece of the code. Basically, this allows us to point to the correct file dynamically. The assumption here is also that the class folder is in the same directory as the current file.

When is the_autoload function called?

The_autoload function is called anytime a reference to an unknown class is made in your code. Of course, the_autoload function must be defined by you in order to actually be called.

Does the_autoload function work with static function calls?

Yes, it does. Remember that a static function is a function that can be called by just using the class name in which it is defined – and there is no need to create an object of the class. The code below, which has a call to a static function, will still run the_autoload function:

```
function __autoload($class_name)
{
      require_once "./classes.".$class_name."-php";
}
//this is a call to a static function
SampleClass::staticFunctionCall($param);
```

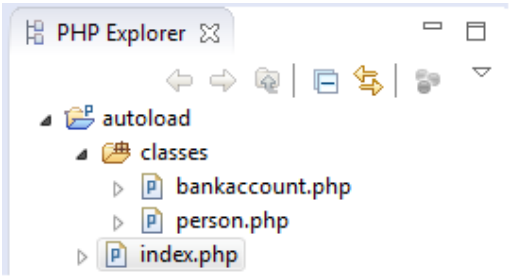
In the code above, the class SampleClass is not recognized because it is not explicitly included anywhere in the code. This means that PHP will make a call to the_autoload function when it realizes that the SampleClass definition is not provided anywhere. Once the__autoload function is called, the class.SampleClass.php file will be included in order to have the definition of the SampleClass class. Of course, the SampleClass is needed because a call is being made to a static function that belongs to the SampleClass class.

When else would the_autoload function be called automatically by PHP?

One last thing that is interesting and worth noting is that even calling the class_exists (which just checks to see if a given class is defined) PHP function will call the_autoload function by default. There is an extra parameter in the class_exists function that would allow you to disable the automatic call to the_autoload function.

Explained Example

You will learn how to organize your class files and load them automatically using PHP_autoload() function.



It is good practice to keep each PHP class in a separated file and all class files in a folder named classes. In addition, the name of the class should be used to name the file for example if you have a Person class, the file name should be person.php.

Before using a class you need to load it in your script file using PHP include file function such as require_once() as follows:

```
require_once("classes/bankaccount.php");
$account = new BankAccount();
```

When the number of classes grows, it is quite tedious to call require_once() function everywhere you use the classes. Fortunately, PHP provides a mechanism that allows you to load class files automatically whenever you try to create an object from a non-existent class in the context of the current script file.

PHP will call_autoload() function automatically whenever you create an object from a non-existent class. To load class files automatically, you need to implement the_autoload() function somewhere in your web application as follows:

```
<?php
function __autoload($className)
{
      $className = str_replace ('..', '', $className);
      require_once ("classes/$className.php");
}
```

Whenever you create an object from a non-existent class, PHP looks into the ./classes folder and load the corresponding class file using the require_once() function.

From now on, you don't have to call require_once() function every time you use the class, which is very useful.

Notice that if PHP cannot find the_autoload() function or if the_autoload() function failed to load the class files, you will get an error message.

Let us see an example of autoloading classes… @ inside the (#15) autoload directory files…

autoload/classes/bankaccount.php

```
class BankAccount
{     function_construct()
      {     echo 'initialize bank account' . '<br/>'; }
}
```

autoload/classes/person.php

```
class Person
{     function_construct()
      {     echo 'initialize person' . '<br/>';       }
}
```

autoload/index.php

```
function __autoload($className)
{
      $className = str_replace('..', '', $className);
      require_once("classes/$className.php");

//    $file_name = "classes/$className.php";
//    if( file_exists ($file_name))
```

```
//    {     require $file_name;     }
}
$account = new BankAccount();
$person = new Person();
```

Class constants

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

Class constants are just like regular constants, except for the fact that they are declared on a class and therefore also accessed through this specific class. Just like with static members, you use the double-colon "::" operator to access a class constant. Here is a basic example: @ 16_class_constant.php

```
class User
{
      const DefaultUsername = "John Doe";
      const MinimumPasswordLength = 6;
}
echo "The default username is " . User::DefaultUsername . "<br>";
echo "The minimum password length is " . User::MinimumPasswordLength;
```

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

Final Keyword

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

A final class could look like this:

```
final class Animal
{     public $name;     }
```

A class with a final function could look like this:

```
class Animal
{     final public function Greet()
      {     return "The final word!";     }
}
```

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

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()
@ 17_final.php

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

```
	// remove final keyword to display the output without Fatal error.
	final public function moreTesting()
	{
		echo "BaseClass::moreTesting() called<br>";
	}
}
class ChildClass extends BaseClass
{
	public function moreTesting()
	{
		BaseClass::moreTesting();
		echo "ChildClass::moreTesting() called<br>";
	}
}
$obj = new ChildClass();
$obj->test();
$obj->moreTesting();
```

MySQL Database handling with OOPs [insert, update, select, delete]

Numerous examples from robots to bicycles have been offered as "easy" explanations of what OOP is. Let's see how OOP works with a real-life example, for a programmer. By creating a MySQL CRUD class you can easily create, read, update and delete entries in any of your projects, regardless of how the database is designed.

Setting up the skeleton of our class is fairly simple once we figure out exactly what we need. First we need to make sure that we can do our basic MySQL functions. In order to do this, we need the following functions:

- Connect
- Disconnect
- Select
- Insert
- Delete
- Update

Those seem pretty basic, but I'm sure that as we go through, we'll notice that a lot of them utilize some similar aspects, so we may have to create more classes. Here is what your class definition should look like. Notice that I made sure that the methods were created with the public keyword.

```
class Database
{
        public function connect()       {    }
        public  function  disconnect()  {  }
        public function select()        {    }
        public function insert()        {    }
        public function delete()        {    }
        public function update()        { }
}
```

Function connect ( )

This function will be fairly basic, but creating it will require us to first create a few variables. Since we want to make sure that they can't be accessed from outside our class, we will be setting them as private. These variables will be used to store the host, username, password and database for the connection. Since they will pretty much remain constant throughout, we don't even need to create modifier or accessor methods for it. After that, we'd just need to create a simple mysql statement to connect to the database. Of course, since as programmers we always have to assume the user (even if it is us) will do something stupid, let's add an extra layer of precaution. We can check if the user has actually connected to the database first, and if they have, there really isn't a need to re-connect. If they haven't then we can use their credentials to connect.

```
private $db_host = '';
private $db_user = '';
private $db_pass = '';
private $db_name = '';

public function connect()
{
        if(!$this->con)
        {
                $myconn = @mysql_connect($this->db_host,$this->db_user,$this->db_pass);
                if($myconn)
                {
                        $seldb = @mysql_select_db($this->db_name,$myconn);
                        if($seldb)
```

```
                {
                        $this->con = true;
                        return true;
                }
                else
                        return false;
        }
        else
                return false;
    }
    else
        return true;
}
```

As you can see, it makes use of some basic mysql functions and a bit of error checking to make sure that things are going according to plan. If it connects to the database successfully it will return true, and if not, it will return false. As an added bonus it will also set the connection variable to true if the connection was successfully complete.

Public function disconnect ( )

This function will simply check our connection variable to see if it is set to true. If it is, that means that it is connected to the database, and our script will disconnect and return true. If not, then there really isn't a need to do anything at all.

```
public function disconnect()
{
    if($this->con)
    {
        if(@mysql_close())
        {
            $this->con = false;
            return true;
        }
        else
        {   return false;   }
    }
}
```

Public function select ( )

This is the first function where things begin to get a little complicated. Now we will be dealing with user arguments and returning the results properly. Since we don't necessarily want to be able to use the results right away we're also going to introduce a new variable called result, which will store the results properly. Apart from that we're also going to create a new function that checks to see if a particular table exists in the database. Since all of our CRUD operations will require this, it makes more sense to create it separately rather than integrating it into the function. In this way, we'll save space in our code and as such, we'll be able to better optimize things later on. Before we go into the actual select statement, here is the tableExists function and the private results variable.

```
private $result = array();
private function tableExists($table)
{
    $tablesInDb = @mysql_query('SHOW TABLES FROM '.$this->db_name.' LIKE "'.$table.'"');
    if($tablesInDb)
    {
        if(mysql_num_rows($tablesInDb) == 1)
        {   return true;    }
        else
```

```
            {       return false;       }
      }
}
```

This function simply checks the database to see if the required table already exists. If it does it will return true and if not, it will return false.

```
public function select($table, $rows = '*', $where = null, $order = null)
{
      $q = 'SELECT '.$rows.' FROM '.$table;
      if($where != null)
            $q .= ' WHERE '.$where;
      if($order != null)
            $q .= ' ORDER BY '.$order;
      if($this->tableExists($table))
      {
            $query = @mysql_query($q);
            if($query)
            {
                  $this->numResults = mysql_num_rows($query);
                  for($i = 0; $i < $this->numResults; $i++)
                  {
                        $r = mysql_fetch_array($query);
                        $key = array_keys($r);
                        for($x = 0; $x < count($key); $x++)
                        {
                              // Sanitizes keys so only alpha values are allowed
                              if(!is_int($key[$x]))
                              {
                                    if(mysql_num_rows($query) > 1)
                                          $this->result[$i][$key[$x]]                 =
$r[$key[$x]];
                        else if(mysql_num_rows($query) < 1)
                            $this->result = null;
                        else
                            $this->result[$key[$x]] = $r[$key[$x]];
                    }
                }
            }
            return true;
        }
        else
        {
            return false;
        }
        }
else
      return false;
   }
```

While it does seem a little scary at first glance, this function really does a whole bunch of things. First off it accepts 4 arguments, 1 of which is required. The table name is the only thing that you need to pass to the function in order to get results back. However, if you want to customize it a bit more, you can do so by adding which rows will be pulled from the database, and you can even add a where and order clause. Of course, as long as you pass the first value, the result will default to their preset ones, so you don't have to worry about setting all of them. The bit of code right after the arguments just serves to compile all our arguments into a select statement. Once that is done, a check is done to see if the table exists, using our

prior tableExists function. If it exists, then the function continues onwards and the query is performed. If not, it will fail.

The next section is the real magic of the code. What it does is gather the columns and data that was requested from the database. It then assigns it to our result variable. However, to make it easier for the end user, instead of auto-incrementing numeric keys, the names of the columns are used. In case you get more than one result each row that is returned is stored with a two dimensional array, with the first key being numerical and auto-incrementing, and the second key being the name of the column. If only one result is returned, then a one dimensional array is created with the keys being the columns. If no results are turned then the result variable is set to null. As I said earlier, it seems a bit confusing, but once you break things down into their individual sections, you can see that they are fairly simple and straightforward.

Public function insert ( )

This function is a lot simpler than our prior one. It simply allows us to insert information into the database. As such we will require an additional argument to the name of the table. We will require a variable that corresponds to the values we wish to input. We can simply separate each value with a comma. Then, all we need to do is quickly check to see if our tableExists, and then build the insert statement by manipulating our arguments to form an insert statement. Then we just run our query.

```
public function insert($table,$values,$rows = null)
    {
        if($this->tableExists($table))
        {
            $insert = 'INSERT INTO '.$table;
            if($rows != null)
            {
                $insert .= ' ('.$rows.')';
            }

            for($i = 0; $i < count($values); $i++)
            {
                if(is_string($values[$i]))
                    $values[$i] = '"'.$values[$i].'"';
            }
            $values = implode(',',$values);
            $insert .= ' VALUES ('.$values.')';
            $ins = @mysql_query($insert);
            if($ins)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
```

As you can see, this function is a lot simpler than our rather complex select statement. Our delete function will actually be even simpler.

public function delete()

This function simply deletes either a table or a row from our database. As such we must pass the table name and an optional where clause. The where clause will let us know if we need to delete a row or the whole table. If the where clause is passed, that means that entries that match will need to be deleted. After we figure all that out, it's just a matter of compiling our delete statement and running the query.

```
public function delete($table,$where = null)
    {
        if($this->tableExists($table))
        {
            if($where == null)
            {
                $delete = 'DELETE '.$table;
            }
            else
            {
                $delete = 'DELETE FROM '.$table.' WHERE '.$where;
            }
            $del = @mysql_query($delete);

            if($del)
            {
                return true;
            }
            else
            {
               return false;
            }
        }
        else
        {
            return false;
        }
    }
```

And finally we get to our last major function. This function simply serves to update a row in the database with some new information. However, because of the slightly more complex nature of it, it will come off as a bit larger and infinitely more confusing. Never fear, it follows much of the same pattern of our previous function. First it will use our arguments to create an update statement. It will then proceed to check the database to make sure that the tableExists.

If it exists, it will simply update the appropriate row. The hard part, of course, comes when we try and create the update statement. Since the update statement has rules for multiple entry updating (IE – different columns in the same row via the cunning use of comma's), we will need to take that into account and create a way to deal with it. I have opted to pass the where clause as a single array. The first element in the array will be the name of the column being updated, and the next will be the value of the column. In this way, every even number (including 0) will be the column name, and every odd number will be the new value. The code for performing this is very simple, and is presented below outside the function:

```
for($i = 0; $i < count($where); $i++)
            {
                if($i%2 != 0)
                {
                    if(is_string($where[$i]))
                    {
                        if(($i+1) != null)
                            $where[$i] = '"'.$where[$i].'" AND ';
                        else
                            $where[$i] = '"'.$where[$i].'"';
                    }
                   else
                    {
                        if(($i+1) != null)
                            $where[$i] = $where[$i]. ' AND ';
```

```
                    else
                        $where[$i] = $where[$i];
                }
            }
        }
```

The next section will create the part of the update statement that deals with actually setting the variables. Since you can change any number of values, I opted to go with an array where the key is the column and the value is the new value of the column. This way we can even do a check to see how many different values were passed to be updated and can add comma's appropriately.

```
$keys = array_keys($rows);
        for($i = 0; $i < count($rows); $i++)
        {
            if(is_string($rows[$keys[$i]]))
            {
                $update .= $keys[$i].'="'.$rows[$keys[$i]].'"';
            }
            else
            {
                $update .= $keys[$i].'='.$rows[$keys[$i]];
            }
            // Parse to add commas
            if($i != count($rows)-1)
            {
                $update .= ',';
            }
        }
```

Now that we've got those two bits of logic out of the way, the rest of the update statement is easy. Here it is presented below:

```
public function update($table,$rows,$where)
    {
        if($this->tableExists($table))
        {
            // Parse the where values
            // even values (including 0) contain the where rows
            // odd values contain the clauses for the row
            for($i = 0; $i < count($where); $i++)
            {
                if($i%2 != 0)
                {
                    if(is_string($where[$i]))
                    {
                        if(($i+1) != null)
                            $where[$i] = '"'.$where[$i].'" AND ';
                        else
                            $where[$i] = '"'.$where[$i].'"';
                    }
                }
            }
            $where = implode('=',$where);

            $update = 'UPDATE '.$table.' SET ';
            $keys = array_keys($rows);
            for($i = 0; $i < count($rows); $i++)
            {
                if(is_string($rows[$keys[$i]]))
                {
```

```
                $update .= $keys[$i].'="'.$rows[$keys[$i]].'"';
            }
            else
            {
                $update .= $keys[$i].'='.$rows[$keys[$i]];
            }

            // Parse to add commas
            if($i != count($rows)-1)
            {
                $update .= ',';
            }
        }
        $update .= ' WHERE '.$where;
        $query = @mysql_query($update);
        if($query)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else
    {
        return false;
    }
}
```

Now that we have that we've finished our last function, our simple CRUD interface for MySQL is complete. You can now create new entries, read specific entries from the database, update entries and delete things. Also, be creating and reusing this class you'll find that you are saving yourself a lot of time and coding. Ah, the beauty of object oriented programming.