PHP Object Oriented Programming

Special thanks to http://www.phpro.org/

What is OOP
Basically, Object Oriented Programming (OOP) is a programming concept that treats functions and data as objects. OK, not the best definition but we've all done it in other classes.

What is an Object
Simply put, an object is a grouping of variables (properties) and functions (methods) all lumped into a single entity. The object can then be called rather than calling the variables or functions themselves.

What is a Class
A class is the blueprint for your object. The class contains the methods and properties, or the characteristics of the object. As an example, let's go with a vehicle. All vehicles share similar characteristics: doors, they are painted some color, they each have a price. All vehicles do similar things: drive, turn left, turn right, stop etc. These can be described as functions, or methods. So, the class holds the definition, and the object holds the values for a particular item. You declare a class in PHP by using the class keyword.

```
class vehicle{
/*** define public properties ***/

/*** the color of the vehicle ***/
public $color;

/*** the number of doors ***/
public $num_doors;

/*** the price of the vehicle ***/
public $price;

/*** the shape of the vehicle ***/
public $shape;

/*** the brand of vehicle ***/
public $brand;

/*** the constructor ***/
public function __construct(){
  echo 'About this Vehicle.<br />';
}

/*** define some public methods ***/

/*** a method to show the vehicle price ***/
public function showPrice(){
  echo 'This vehicle costs '.$this->price.'.<br />';
}

/*** a method to show the number of doors ***/
public function numDoors(){
  echo 'This vehicle has '.$this->num_doors.' doors.<br />';
}

/*** method to drive the vehicle ***/
public function drive(){
```

```
    echo 'VRRROOOOOOM!!!';
  }

  } /*** end of class ***/
```

With this simple class definition we can now create one, or many, vehicle objects. Let's see how it's done, then we will step through it all. To create a new object from the class definition we use the new keyword.

```
/*** create a new vehicle object ***/
$vehicle = new vehicle;

/*** the brand of vehicle ***/
$vehicle->brand = 'Porsche';

/*** the shape of vehicle ***/
$vehicle->shape = 'Coupe';

/*** the color of the vehicle ***/
$vehicle->color = 'Red';

/*** number of doors ***/
$vehicle->num_doors = 2;

/*** cost of the vehicle ***/
$vehicle->price = 100000;

/*** call the showPrice method ***/
$vehicle->showPrice();

/*** call the numDoors method ***/
$vehicle->numDoors();

/*** drive the vehicle ***/
$vehicle->drive();
```

Explanation… We began the class with the class keyword. This tells PHP that what follows is a class definition.

```
class vehicle {
```

Next we declared some variables, or as they are known in the OOP world, properties.

```
public $color;
public $num_doors;
public $price;
public $shape;
public $brand;
```

The properties are the individual characteristics of the vehicle object. The public declares that this property may be accessed within the public scope. This means you may set it from within your code. Next in the class definition is this code.

```
public function __construct(){
  echo 'About this vehicle.<br />';
}
```

The constructor is a special function that is executed every time a new object is created or "instantiated." If a class needs to do something before running some other piece of code, such as establishing member variable values or connecting to a database, then this is the place to do it. Following the constructor are several

methods that perform actions. These methods have been set to public meaning they can be called from anywhere the object exists.

```
public function showPrice(){
  echo 'This vehicle costs '.$this->price.'.<br />';
}

public function numDoors(){
  echo 'This vehicle has '.$this->num_doors.' doors.<br />';
}

public function drive(){
  echo 'VRRROOOOOOM';
}
```

As you see above, our methods are similar to normal PHP functions. However you may note the use of variables like $this->price and $this->num_doors. The keyword $this is used to refer to properties or methods within the class itself. The $this keyword is reserved in PHP, so it cannot be used as a variable or property name.

After designing or copying over a class, you can begin to use or instantiate objects of your class. As mentioned earlier we do this with the use of the new keyword.

```
/*** create a new vehicle object ***/
$vehicle = new vehicle;
```

The code above creates a new object from our class definition called $vehicle. It is at this time that the constructor is called and any code within the constructor is executed. Once we have a new object we can then begin to assign values to the properties of that object as seen here.

```
/*** the brand of vehicle ***/
$vehicle->brand = 'Porsche';

/*** the shape of vehicle ***/
$vehicle->shape = 'Coupe';

/*** the color of the vehicle ***/
$vehicle->color = 'Red';

/*** number of doors ***/
$vehicle->num_doors = 2;
```

With the properties set as above, it is now simple to use the object methods to manipulate the object by calling the methods. The methods are executed like this below.

```
/*** call the showPrice method ***/
$vehicle->showPrice();

/*** call the numDoors method ***/
$vehicle->numDoors();

/*** drive the vehicle ***/
$vehicle->drive();
```

Inheritance (Extending a class)

Probably the greatest feature of the PHP OOP model is Inheritance. Inheritance is the ability of PHP to extend classes (child classes) that inherit the characteristics of the parent class. The resulting object of an extended

class has all the characteristics of the parent class, plus whatever is in the new child, or extended class. In this instance we will extend the vehicle class and add a motorcycle. A motorcycle is still a vehicle and shares many of the same attributes such as price, etc., but a motorcycle has some unique features that a car does not. Rather than type out the whole of the vehicle class definition again, we will save it in a file of its own called vehicle.class.php. It is important here to note the naming convention here as it will be important later in this tutorial. Lets see how it works.

```
/*** include the vehicle class definition ***/
include('vehicle.class.php');

class motorcycle extends vehicle{

/*** the number of side cars ***/
private $num_sidecars;

private $handlebars;

/**
*
* set the type of handlebars
*
* @access public
*
* @param string
*
* @return string
*
**/
public function setHandlebars($handlebarType){
  $this->handlebars=$handlebarType;
}

/**
*
* Set number of side cars
*
* @access public
*
* @param int
*
* @return int
*
**/
public function setSidecar($numSidecars){
  $this->numSidecars = $numSidecars;
}


/**
*
* Show the numbers of sidecars
*
* @return string
*
**/
public function showSideCar(){
  echo 'This motorcyle has '. $this->numSidecars.' sidecar<br />';
}

} /*** end of class ***/
```

```php
/*** our code ***/

/*** create a new vehicle object ***/
$vehicle = new motorcycle;

/*** the brand of vehicle ***/
$vehicle->brand = 'Harley Davidson';

/*** the shape of vehicle ***/
$vehicle->shape = 'Sportster';

/*** the color of the vehicle ***/
$vehicle->color = 'Black';

/*** number of doors ***/
$vehicle->num_doors = 0;

/*** cost of the vehicle ***/
$vehicle->price = 25000;

/*** type of handle bars ***/
$vehicle->setHandlebars('Ape Hangers');

/*** set the sidecar ***/
$vehicle->setSidecar(1);

/*** show the vehicle brand and type ***/
echo $vehicle->brand.': '.$vehicle->shape.'<br />';


/*** call the showPrice method ***/
$vehicle->showPrice();

/*** show the sidecars ***/
$vehicle->showSideCar();

/*** drive the vehicle ***/
$vehicle->drive();
```

You see in the motorcycle class that we have used the keyword extends to extend our vehicle class. We can then proceed to set vars in the both the parent class and the child class. The child class has inherited all the characteristics of the parent vehicle class.

If you run the above code with out including the parent class, you will receive a Fatal Error.

Visibility (public, private, protected)

The visibility of properties and methods relates to how that member may be manipulated within, or from outside the class. Three levels of visibility exist for PHP class members, just like C++.

- public
- private
- protected

One major difference is, by default, all class members are public. This means if you do not declare a property or method within the class, it will be public. It is good practice to declare the visibility of class members for the sake of readability for yourself and others. It is much easier for another programmer to see your intentions.

Public class members are available throughout the script and may be accessed from outside the class as we demonstrated in our car class.

```
class mathematics{
/*** a  number ***/
public $num;

/**
*
* Add two to a number
*
* @access public
*
* @return int
*
**/
public function addTwo(){
  return $this->num+2;
}

}/*** end of class ***/

/*** Instantiate a new class instance ***/
$math = new mathematics;

/*** set the value of the number ***/
$math->num = 2;

/*** call the addTwo method ***/
echo $math->addTwo();
```

We can see in the above example that the public variable $num is set from user space and we call a public method that adds two the number and returns the value of $num+2. Having our properties (variables) visible, or accessible from any part of our script works in our favor here, but it can also work against us. A problem could arise if we lost track of our values and changed the value of $num. To counter this problem we can create a method to set the value for us. Even with this in place it is still possible for somebody to simply access the $num variable. So we make the variable private. This ensures us that the property is only available within the class itself. It is private to the calling class. Consider the following code.

```
class mathematics{

/*** a  number ***/
private $num;

/**
*
* Set the value of $num
*
* @access public
*
* @param $num The number to set
*
* @return int
*
**/
public function setNum($num){
  $this->num = $num;
}

/**
```

```
*
* Add two to a number
*
* @access public
*
* @return int
*
**/
public function addTwo(){
  return $this->num+2;
}

}/*** end of class ***/

/*** Instantiate a new class instance ***/
$math = new mathematics;

/*** set the value of the number ***/
$math->setNum(2);

/*** call the addTwo method ***/
echo $math->addTwo();
```

Any further attempt to reset the $num property without the setNum() method would result in a Fatal Error

Even if you were to try to access the private $num property from a child class it would fail. This is because private methods and properties in a parent class are not visible to child classes and cannot be accessed. To access a parent method or property from a child class you need to use the protected keyword. Like the private keyword, protected methods and properties are available only to the class that created them. But unlike private, protected methods and properties are visible from a parent class. Lets see how this works.

```
class mathematics{
/*** a  number ***/
protected $num;

/**
*
* Set the value of $num
*
* @access public
*
* @param $num The number to set
*
* @return int
*
**/
public function setNum($num){
  $this->num = $num;
}

/**
*
* Add two to a number
*
* @access public
*
* @return int
*
**/
public function addTwo(){
  return $this->num+2;
```

```
}

}/*** end of class ***/

class divide extends mathematics{

/**
*
* Divide a number by two
*
* @access public
*
* @return int
*
**/
public function divideByTwo(){
  /*** divide number by two ***/
  $new_num = $this->num/2;
  /*** return the new number and round to 2 decimal places ***/
  return round($new_num, 2);
}

} /*** end of class ***/

/*** Instantiate a new class instance ***/
$math = new divide;

/*** set the value of the number ***/
$math->setNum(14);

echo $math->divideByTwo();
```

We can see here the the user space code has called the setNum() method in the parent mathematics class. This method, in turn, sets the $num property. We are able to do this because the $num property has been declared protected and is visible to the child class.

Final

As we saw in the previous section there are ways to protect your code from being used in an improper manner. Another way of protecting yourself is the Final keyword. Any method or class that is declared as Final cannot be overridden or inherited by another class.

```
final class mathematics{

} /*** end of class ***/

class divide extends mathematics{

}
```

By running the above code you will get an error. This can protect us from those who wish to use our code for a purpose other than that for which it was intended.

Abstract Classes

An abstract class is a class that cannot be instantiated on its own. You cannot create a new object from it. To see this lets make a basic class.

```
abstract class mathematics{

/**
*
* Add two to a number
*
* @access public
*
* @return int
*
**/
public function addTwo(){
  return $this->num+2;
}

} /*** end of class ***/

/*** try to create new object from the mathematics class ***/
$math = new mathematics
```

Using the code above, you will get an error. As you can see, this is not allowed. Also if you declare any class method to be abstract, you must also declare the class itself as abstract too. So, what's the point you ask? Well, you can inherit from an abstract class. Any class that extends an abstract parent class must create an interface of the parent abstract methods. If this is not done a fatal error is generated. This ensures that the implementation is correct for your application.

```
abstract class mathematics{
/*** child class must define these methods ***/
abstract protected function getMessage();

abstract protected function addTwo($num1);

/**
*
* method common to both classes
*
**/
public function showMessage() {
  echo $this->getMessage();
}

} /*** end of class ***/

class myMath extends mathematics{
/**
*
* Prefix to the answer
*
* @return string
*
**/
protected function getMessage(){
  return "The anwser is: ";
}

/**
*
* add two to a number
*
* @access public
*
```

```
* @param $num1 A number to be added to
*
* @return int
*
**/
public function addTwo($num1) {
  return $num1+2;
}

} /*** end of class ***/

/*** a new instance of myMath ***/
$myMath = new myMath;
/*** show the message ***/
$myMath->showMessage();
/*** do the math ***/
echo $myMath->addTwo(4);
```

Static Methods and Properties

The use of the static keyword allows class members to be used without needing to instantiate a new instance of the class. The static declaration must come after the visibility declaration, eg: public static myClass{

Because there is no object created when using a static call, the keyword $this and the arrow operator, -> are not available. Static variables belong to the class itself and not to any object of that class. To access within the class itself you need to use the self keyword along with the :: scope resolution operator.

```
/*** a simple class ***/
class myClass{
/*** a static variable ***/
static $foo;
}

/** give the static variable a value ***/
myClass::$foo = 'Bar';

/*** echo the static variable ***/
echo (myClass::$foo ).
```

This is rather basic as an example, so lets use something practical. Static properties are often used as counters. Here we will use a basic counter class.

```
class counter{

/*** our count variable ***/
private static $count = 0;

/**
* Constructor, duh
**/
function __construct() {
  self::$count++;
}

/**
*
* get the current count
*
* @access public
*
```

```
    * @return int
    *
    **/
    public static function getCount() {
     return self::$count;
    }

    } /*** end of class ***/

    /*** create a new instance ***/
    $count = new counter();
    /*** get the count ***/
    echo counter::getCount() . '<br />';
    /*** create another instance ***/
    $next = new counter();
    /*** echo the new count ***/
    echo counter::getCount().'<br />';
    /*** and a third instance ***/
    $third = new counter;
    echo counter::getCount().'<br />';
```

At each new instance the counter class we increment by one. Not also the use of the :: scope resolution operator and self keyword to refer to the static variable within the class itself.

PHP Class functions

PHP has available several class functions to help you through the OOP mine field.

- class_exists()
- get_class()
- get_declared_classes()

```
/*** a pretend class ***/
class fax{
  public function dial(){ }
  public function send(){ }
  public function recieve(){ }
}

/*** another pretend class ***/
class printer{
  public function printBlack(){ }
  public function printColor(){ }
  public function printDraft(){ }
  public function kick(){ }
}

/*** create an instance of the fax class ***/
$foo = new fax;

/*** create an instance of the printer class ***/
$bar = new printer;

echo '$foo is from the ' get_class($foo).' class<br />';
echo class_exists("printer").'<br />';
echo 'Declared classes are:<br /> ';
foreach(get_declared_classes() as $key=>$classname)
    {
    echo $key.' -&gt; '.$classname.'<br />';
    }
```

The little snippet above will produce over one hundred available classes, shortened here for the sake of sanity, such as these below. Note our fax and printer classes at the bottom of the list.

- $foo is from the fax class
- 1
- 0 -> stdClass
- 1 -> Exception
- ---8<--- snip ---
- 106 -> fax
- 107 -> printer

Autoload

Earlier in this tutorial we stated that the class definition must be included in every call to an object. This is commonly achieved with the include() or require() functions such as below.

```
/*** include our class definitions ***/
include('classes/vehicle.class.php');

include('classes/motorcycle.class.php');

include('classes/printer.class.php');

include('classes/printer.class.php');

/*** instantiate a new vehicle class object ***/
$vehicle = new vehicle;

*** instantiate a new motorcycle class object ***/
$bike = new motorcycle;

*** instantiate a new printer class object ***/
$printer = new printer;

*** instantiate a new fax class object ***/
$fax = new fax;
```

As you can see, this is quite cumbersome. The solution to this sort of mess is __autoload(). The __autoload() function will internally search out the class and load its definition. So the above block of code could be reduced to this.

```
/*** Autoload class files ***/
function __autoload($class){
  require('classes/' . strtolower($class) . '.class.php');
}

/*** instantiate a new vehicle class object ***/
$vehicle = new vehicle;

/*** instantiate a new motorcycle class object ***/
$bike = new motorcycle;

/*** instantiate a new printer class object ***/
$printer = new printer;

/*** instantiate a new fax class object ***/
$fax = new fax;
```

Now we can load up as many class definitions as we like because they will be autoloaded when we try to use a class that has not been defined. This can save much coding and much searching for code. It is important to remember the naming convention of your classes and class files. Each class file should be named the same as the class definition itself. eg: a class definition file named fax would have the filename fax.class.php
The use of the strtolower() function assures compatibility of naming conventions as windows machines fail to be case sensitive for filenames.

Serializing Objects

We have seen a lot of code above for the use of objects and how they can save us time (and $$$) by re-using them. But what if we needed to somehow store an object for later retrieval, perhaps in a database or in a session variable? PHP has given us the serialize() function to make this rather effortless. There are some limitations, but this can be a very useful tool for applications.

Overloading

Overloading in PHP has caused much confusion for no real reason. PHP Overloading can be broken down into two basic components

- Method overloading
- Property overloading

Simply put, Method Overloading is achieved by a special function named __call(). It is available as a sort of method wildcard for calls to undefined methods within a class. The __call() will only work when the class method you are trying to access does not exist.

```
class my_class{
 public function foo() {
    return "This is the foo function";
 }

} /*** end of class ***/

/*** create a new class object ***/
$obj = new my_class;
/*** call a non-existant method ***/
$obj->bar();
```

Of course the above snippet of code will produce an error such as

With the __call() function in place, PHP will try to create the function and excecute any code within the _call() method. The __call() method takes two arguments, the method name, and the arguments passed to the function call. Your call to the undefined method may have many arguments and these are returned in an array.

```
class my_class{
 public function foo() {
    return "This is the foo function";
 }
  /*** __call() method with two args ***/
  public function __call($method, $arg){
   echo $method.'<br />';
   print_r($arg);
 }
} /*** end of class ***/
```

```
/*** create a new class object ***/
$obj = new my_class;
/*** call a non-existant method ***/
$obj->bar("arg1", "arg2");
?>
```

The __call() method has returned the method name that we called along with the array of args passed to it. Let's now look at we can dynamically manipulate or overload our data.

```
class readyGetSet {
/*** declare $item ***/
private $item = 'Skate Board';
/*** declare the price ***/
private $price = 100;

/*** our call function ***/
function __call($method, $arguments){
/*** set property and prefix ***/
 $prefix   = strtolower(substr($method, 0, 3));
 $property = strtolower(substr($method, 3));

 if (empty($prefix) || empty($property))
        {
        return;
        }

if ($prefix == 'get' && isset($this->$property))
        {
        return $this->$property;
        }

if ($prefix == 'set')
        {
        $this->$property = $arguments[0];
        }
    }
}

$obj = new readyGetSet;

echo 'Item: ' . $obj->getItem() . '<br />';
echo 'Price: ' . $obj->getPrice() . '<br />';

$obj->setItem('CD');
$obj->setPrice(25);

echo 'Item: ' . $obj->getItem() . '<br />';
echo 'Price: ' . $obj->getPrice() . '<br />';
```

The second part of overloading refers to properties and the ability to be able to dynamically get and set object properties. The __get() function is called when reading the value of an undefined property, and __set() is called when trying to change that properties value.

```
 class candy{
 /*** declare a property ***/
 public $type='chocolate';

 /*** a normal method ***/
 public function wrap(){
   echo 'Its a wrap';
 }
```

```
 /*** our __set() function ***/
 public function __set($index, $value){
  echo 'The value of '.$index.' is '.$value;
 }

} /*** end of class ***/

/*** a new candy object ***/
$candy = new candy;
/*** set a non existant property ***/
$candy->bar = 'Blue Smarties';
```

We have described a class named candy which contains a public property named $type. It has a simple method and our __set() method. After the class our user code creates a new instance of the candy class. Then we try to set a variable that does not exist in the class. Here the __set method takes control and assigns it for us. We then see in our __set method that it echoes the name of the variable, plus its intended value. The __set() method takes two arguements, the name of the non existant variable, and its intended value.
The __get() method ....

```
 class candy{
 /*** declare a property ***/
 public $type='chocolate';

 public $choctype = array('milk'=>0, 'dark'=>1, 'plain'=>2);

 /*** a normal method ***/
 public function wrap(){
   echo 'Its a wrap';
 }
 /*** our __set() function ***/
 public function __get($index){
    echo 'Retrieving element of $choctype property with index of '.$index.'<br />';
     return $this->choctype[$index];
 }

} /*** end of class ***/

/*** a new candy object ***/
$candy = new candy;
/*** set a non existant property ***/
 echo 'The value of the following element property is '.$candy->milk;
```

Class Constants

You have more than likely seen the use standard constants in PHP. To define a standard constant we use this code:

```
/*** define an error message ***/
define('_ERROR_MSG', 'An Error has occured!');
/*** echo the constant ***/
echo _ERROR_MSG;
```

To define a class constant we use the const keyword.

```
class my_class {
 /*** define a class constant ***/
 const _ERROR_MSG = 'An Error has occured!';

public function show(){
 echo self::_ERROR_MSG;
```

```
}

} /*** end of class ***/
```

There are now three ways the class constant can be access from this example.

```
/*** static call to constant ***/
echo my_class::_ERROR_MSG;
/*** instantiate a class object ***/
$my_class = new my_class;
/*** can run the show() method ***/
$my_class->show();
/*** static call to the show method() ***/
my_class::show();
```

A class constant, like standard constants, must be exactly as the name suggests, a constant value. It cannot be a variable or the result of a function or method.

Real World Example – Creating a MySQL Database Wrapper Class for MySQL Functions

```
class Database
{
    /*    * Edit the following variables    */
    private $db_host = 'localhost';       // Database Host
    private $db_user = 'root';            // Username
    private $db_pass = 'root';            // Password
    private $db_name = 'blog';            // Database
    /*     * End edit     */

    private $con = false;               // Checks to see if the connection is active
    private $result = array();          // Results that are returned from the query

    /*    Connects to the database, only one connection allowed   */
    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;
```

```
    }
}

/*   Changes the new database, sets all current results to null  */
public function setDatabase($name)
{
    if($this->con)
    {
        if(@mysql_close())
        {
            $this->con = false;
            $this->results = null;
            $this->db_name = $name;
            $this->connect();
        }
    }
}

/*  Checks to see if the table exists when performing queries  */
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;
        }
    }
}

/* Selects information from the database.
*  Required: table (the name of the table)
*  Optional: rows (the columns requested, separated by commas)
*            where (column = value as a string)
*            order (column DIRECTION as a string)
*/
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;

    $query = @mysql_query($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); $x++)
            {
                // Sanitizes keys so only alphavalues are allowed
                if(!is_int($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;
    }
}

/*  Insert values into the table
*   Required: table (the name of the table)
*           values (the values to be inserted)
*   Optional: rows (if values don't match the number of rows)
*/
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;
        }
    }
}

/*  Deletes table or records where condition is true
*   Required: table (the name of the table)
*   Optional: where (condition [column =  value])
*/
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;
    }
}

/*  Updates the database with the values sent
 *  Required: table (the name of the table to be updated
 *            rows (the rows/values in a key/value array
 *            where (the row/condition in an array (row,condition) )
 */
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;
        }
    }

    /*  Returns the result set  */
    public function getResult()
    {
        return $this->result;
    }
}
```