Using PHP Objects to access your Database Tables (Part 1)

Posted on 31st May 2003 **by Tony Marston**

By Tony Marston

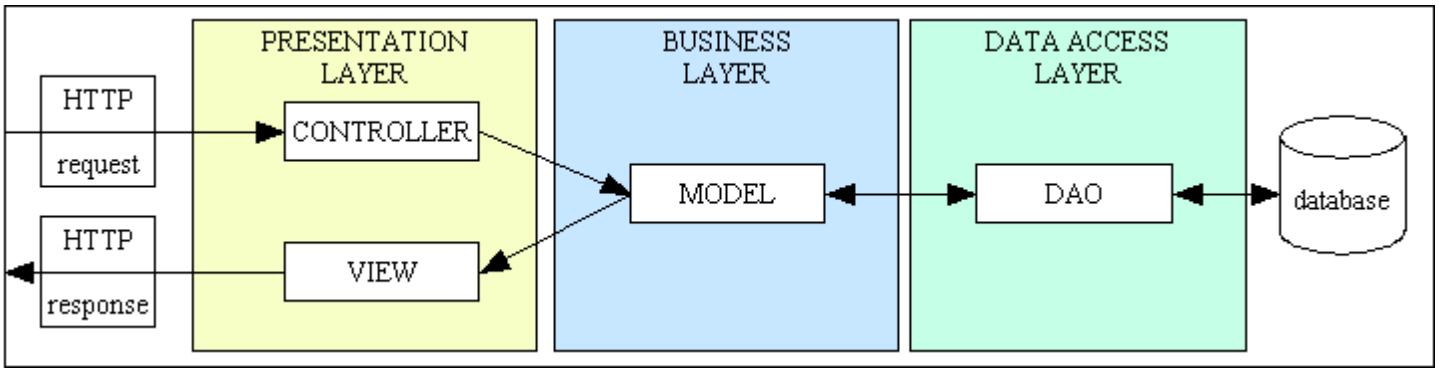
Intended Audience

This tutorial is intended for developers who have already written code to get data in and out of a MySQL database, but who wish to discover if there are any benefits of adopting an Object Oriented approach. This tutorial will show you how to create an abstract class which can deal with any database table, and then how to create subclasses containing the implementation details for each individual table within your application. The end result is that you never have to code any of the SQL SELECT, INSERT, UPDATE or DELETE statements for any table as they will be generated at runtime.

Each of these subclasses can then be considered as the Model component in the Model-View-Controller (MVC) design pattern as they handle the data validation and business rules for their assigned tables. The MVC pattern does not include a component which is specifically designed for accessing a database, so it is sometimes assumed that this is handled within the Model as well. As you progress through this tutorial you will see the advantages of splitting this responsibility off to a separate fourth component known as a Data Access Object (DAO). This will allow you to have one Model component for each database table and one DAO for the DBMS engine. This approach will make it simpler to switch from one DBMS engine to another, such as from the "original" to the "improved" MySQL extension, or even to another DBMS altogether.

Figure1 shows how these four components fit together. Note also that this arrangement also provides an implementation of the 3-Tier Architecture which I encountered in the language which I used before switching to PHP.

Figure 1 - MVC plus 3-Tier Architecture



There are quite a few so-called OOP 'experts' who seem to think that having a separate class for each table in the database is a silly idea, so let me explain the logic in my approach. If I am designing an application to deal with such real-world entities as 'customer', 'product' and 'invoice' then I will want a software module/component/object to deal with each of these entities. This software module will contain (encapsulate) all the information required to process the entity, such as the business rules, and will also be required to read from and write to the database table. In my previous language the word 'entity' was synonymous with the word 'table', so when I talk about creating a class for a database table I actually mean a class for an entity. This includes, but is not limited to, the ability to communicate with the database table associated with that entity. So, entity=table and entity=class, therefore class=table. Simple.

I have to admit up front that I do not come from an OOP (Object Oriented Programming) background. In fact PHP is the first language I have used that has had OOP capabilities. Now there are some people who argue that PHP is not a 'proper' OO language, but they are just nit-picking. While it is true that PHP was not originally designed as an OOP language, and that some more advanced features will not be available until PHP 5 is released, I have found the capabilities of PHP 4 more than adequate for my purposes.

In this tutorial you will learn the following:

- How to define a class with properties and methods.
- How to create an object (an instance of) a class.
- How to create a class which extends another class.
- How to communicate with an object from within a PHP script.

Prerequisites

It is assumed that you are already familiar with the fundamentals of PHP, such as array handling and accessing a database in order to insert, read, update and delete data. Knowledge of PHP 4's object oriented capabilities is not essential as this tutorial will take you through the basics.

An introduction to OO functionality within PHP

Here is a brief overview of the Object Oriented functionality that is available within PHP 4. It is not intended to cover all the possibilities, just the essentials to get you going.

Creating a Class with Properties and Methods

A class can be created using code similar to the following:

```
class Foo
{
   var $foo;
   var $bar;

   function setBar ($bar)
   {
      $this->bar = $bar;
   }
   function getBar ()
   {
      return $this->bar;
   }
}
```

This class has the following characteristics:

- The name of the class is 'Foo'.

- It contains the variables (properties) '$foo' and '$bar'.
- It contains functions (methods) called 'setBar' and 'getBar'.
- Function 'setBar' is used to insert data into the object variable '$bar'.
- Function 'getBar' is used to retrieve data from the object variable '$bar'.

In theory you are supposed to have a 'set' method and a 'get' method for each variable within the class, one to put data in and the other to get data out. These are commonly referred to as 'setters' and 'getters'.

Note here that the syntax `$this->bar` is used to reference an object variable from within that object. These variables can be referenced by any function/method within the class. The syntax `$bar` identifies a variable whose scope is limited to the current function only.

The 'constructor' method

A class can contain a special method known as a 'constructor' which is automatically processed whenever an instance of that class is created. In PHP 4 this is a method which has the same name as the class. This can be used to set initial data for the object, as shown in the following example:

```
class Foo
{
   var $foo;
   var $bar;

   function foo ()
   {
      $this->foo = 'initial value for $foo';
      $this->bar = 'initial value for $bar';
   }
}
```

In PHP 5 you can also use the standard name `__construct()` as the constructor. In PHP 7 you can only use the standard name `__construct()`.

Extending a Class

It is possible to create a new class which 'extends' an existing class. By this I mean that you can inherit all the properties and methods of the existing class, and either provide alternative code for existing methods or add completely new methods. An example of how to do this is shown below:

```
require_once 'foo.class.inc';
class Bar extends Foo
{
   var $tom;
   var $dick;
   var $harry;

   function setTom ($tom)
   {
      $this->tom = $tom;
   }
   function getTom ()
   {
      return $this->tom;
   }
}
```

Note that you have to include the definition of the parent class before you can extend it.

Class `Bar` is now an extension of class `Foo`. It has the following characteristics:

- It has all the properties and methods of class `Foo` plus some properties and methods of its own.
- If class `Bar` contained a method with the same name as a method within class `Foo` then the `Bar` method would replace the `Foo` method.
- If class `Bar` does not have a constructor of its own then the constructor in class `Foo` will be used instead.

Creating an Object

Now that we have created a class how do we use it? The first step is to create an instance of the class known as an object. Note that you must include the definition of the class before you can create an object from that class, as shown below:

```
include 'foo.class.inc';
$object = new Foo;
```

Here the object is called '$object', but I could have used any name. Note that it is possible to create more than one object from the same class:

```
include 'foo.class.inc';
$tom = new Foo;
$dick = new Foo;
$harry = new Foo;
```

Accessing an Object's Properties and Methods

In order to perform a method within an object you need to specify both the object name and the function name as in:

```
$result = $tom->setFoo('value');
```

It is also possible to access an object's properties directly without going through a method, as in:

```
$var = $tom->Foo;
$tom->Foo = $var;
```

Although this approach is perfectly valid I should point out that if at some time in the future you decide that it is necessary to do some extra processing on the data before it is moved in or out of your object then you will have to modify all those places where the data is referenced. On the other hand if you force all object properties to be accessed though a `get` or `set` method then you will only have to change the contents of that method just the once.

You may have noticed that when you are outside of an object and you want to access the object's properties or methods you must specify the object's identifier as in `$tom->` or `$dick->` or `$harry->`, but when you are inside an object you can use the magic word `$this->` as the object identifier.

Objects and Sessions

You may or may not be aware that you can maintain data between the execution of one script and another by using PHP's session capability. It is also possible to save an object's properties in this session data so that it can be reinstated by the next script within the same session. You can save an object's properties by using the `serialize()` command as follows:

```
include 'foo.class.inc';
$dbobject = new Foo;
   ...
   ...
$_SESSION['dbobject'] = serialize($dbobject);
```

In a subsequent script you can reinstate the object to exactly the same condition by using the `unserialize()` command like this:

```
include 'foo.class.inc';
if (isset($_SESSION['dbobject'])) {
   $dbobject = unserialize($_SESSION['dbobject']);
} else {
   $dbobject = new Foo;
} // if
```

My 'database_table' class

Background

I have modeled my approach on the design I used in a language prior to switching to PHP. This prior language was based on components rather than objects, but while reading up on the basics of OOP I discovered that there were in fact some similarities:

Object Oriented	Component Based
You can define CLASSES.	You can define COMPONENTS.
You can define properties (data) which can be maintained within the class.	You can define variables within the component definition.
You can define methods (functions) to manipulate the properties within the class.	You can define operations (functions) within the component definition.

You can define public methods which are accessible from outside the class.	You can define operations which are accessible from outside the component.
You can define private methods which are only accessible from inside the class.	You can define local procedures which are only accessible from inside the component.
You can define a class constructor which is processed when a class instance is created.	You can define an INIT operation which is processed when a component instance is created.
You can define a class destructor which is processed when a class instance is terminated.	You can define a QUIT operation which is processed when a component instance is terminated.
You can create one or more instances of a class, each with its own object name.	You can create one or more instances of a component, each with its own instance name.
A class instance remains in existence until it is terminated.	A component instance remains in existence until it is terminated.
You can access object properties either directly or via a method (although it is considered bad practice to access properties directly).	You can only access component variables via an operation.
You can define a class which extends an existing class, thus inheriting all the properties and methods of that class.	You cannot extend a component, but by using component templates and include files it is possible to share quantities of common code.

While reading what other developers had done with database objects in PHP (and even Java) I noticed several characteristics which I did not have in my previous component-based solution and which I most certainly did not want to have in my new object-based solution. These were:

- Having 'setters' and 'getters' for each column within the table. I do not bother with individual column names as the argument for my standard 'getData' and 'putData' functions is an associative array. As this array can contain any number of 'name=value' pairs I can access the data for any individual column by using PHP's standard array functionality. In my component-based solution I used XML streams instead of arrays, but the principle was the same.

- Having separate instances for each database row. This is total overkill as far as I am concerned. As arrays in PHP can contain separate entries for each row I can use a single object to handle all the rows I need. In my component-based solution the XML stream could also contain any number of rows, so I did not need multiple instances for multiple rows.

- Having method/operation/function names which are specific to the database table being accessed. For example for the CUSTOMER table you would have a 'getCustomer' and 'putCustomer' method while for the PRODUCT table you would have 'getProduct' and 'putProduct'. This immediately means that you cannot use a general-purpose script to communicate with a database object as you would need to know what methods to use instead of the generic 'getData' and 'putData'. Using generic methods for all derived classes actually conforms to the OOP idea of polymorphism, so I do not see why some developers insist on having different methods for each derived class.

The main reason for adopting the OOP approach is to maximise the amount of reusable code, so I set out to create a base class which could contain all the standard code for getting data in and out of any database table. I would then be able to create a separate class for each physical database table which would extend this base class and would therefore only have to contain extra code that is specific to that particular database table.

Class Variables

The first task is the define the class and its variables, like so:

```
class Default_Table
{
  var $tablename;         // table name
  var $dbname;            // database name
  var $rows_per_page;     // used in pagination
  var $pageno;            // current page number
  var $lastpage;          // highest page number
  var $fieldlist;         // list of fields in this table
  var $data_array;        // data from the database
  var $errors;            // array of error messages
```

Class Constructor

This is immediately followed by the constructor method. Note that each derived class should have its own constructor containing proper values:

```
  function Default_Table ()
  {
    $this->tablename       = 'default';
    $this->dbname          = 'default';
    $this->rows_per_page   = 10;

    $this->fieldlist = array('column1', 'column2', 'column3');
    $this->fieldlist['column1'] = array('pkey' => 'y');
  } // constructor
```

You should notice here that the constructor for each table identifies the name of the database to which that particular table belongs. It is therefore possible to create classes for tables which belong to more than one database, and to access more than one database within the same session.

The variable `$fieldlist` is used to list all the columns within that table, and to identify which is the primary key. How this is used will become apparent later on. In the `Default_Table` class this is a dummy array which is replaced with proper values within each database table subclass.

'getData' Method

This is my standard method for getting data out of the database through the object. It can be used to retrieve any number of rows. I start by defining the function name with any arguments, then initialise some variables. Note that `$this->pageno` may have been set previously to request a particular page in a multi-page display. By default this starts at 1, but different values may be requested from the user by using hyperlinks provided on the HTML page.

Someone once suggested that I have a `getNextPage()` and `getPreviousPage()` method to provide the navigation mechanism, but this is both unnecessary and restrictive - my single method can jump to any page that is available rather than current+1 or current-1.

```
   function getData ($where)
   {
      $this->data_array = array();
      $pageno           = $this->pageno;
      $rows_per_page    = $this->rows_per_page;
      $this->numrows    = 0;
      $this->lastpage   = 0;
```

Next I connect to the database using my standard db_connect procedure. Note that the table name is picked up from the variable which was set in the class constructor. In the event on an error this will invoke my standard error handler.

```
      global $dbconnect, $query;
      $dbconnect = db_connect($this->dbname) or trigger_error("SQL", E_USER_ERROR);
```

The input argument `$where` can either be empty or it can contain selection criteria in any of the following formats:

```
column1='value'
column1='value' AND column2='value'
(column1='value' AND column2='value') OR (column1='value' AND column2='value')
```

If `$where` is not empty I construct a separate string to include in any database query.

```
      if (empty($where)) {
         $where_str = NULL;
      } else {
         $where_str = "WHERE $where";
      } // if
```

Next we want to count the number of rows which satisfy the current selection criteria:

```
      $query = "SELECT count(*) FROM $this->tablename $where_str";
      $result = mysql_query($query, $dbconnect) or trigger_error("SQL", E_USER_ERROR);
      $query_data = mysql_fetch_row($result);
      $this->numrows = $query_data[0];
```

If there is no data we can exit at this point.

```
      if ($this->numrows <= 0) {
         $this->pageno = 0;
         return;
      } // if
```

If there is data then we want to calculate how many pages it will take based on the page size given in `$rows_per_page`.

```
      if ($rows_per_page > 0) {
         $this->lastpage = ceil($this->numrows/$rows_per_page);
      } else {
         $this->lastpage = 1;
      } // if
```

Next we must ensure that the requested page number is within range. Note that the default is to start at page 1.

```
      if ($pageno == '' OR $pageno <= '1') {
         $pageno = 1;
      } elseif ($pageno > $this->lastpage) {
         $pageno = $this->lastpage;
      } // if
      $this->pageno = $pageno;
```

Now we can construct the `LIMIT` clause for the database query in order to retrieve only those rows which fall within the specified page number:

```
      if ($rows_per_page > 0) {
         $limit_str = 'LIMIT ' .($pageno - 1) * $rows_per_page .',' .$rows_per_page;
      } else {
         $limit_str = NULL;
      } // if
```

Now we can build the query string and run it.

```
      $query = "SELECT * FROM $this->tablename $where_str $limit_str";
      $result = mysql_query($query, $dbconnect) or trigger_error("SQL", E_USER_ERROR);
```

At this point `$result` is simply a resource that points to the data, so we need to extract the data and convert it into an associative array. This will have an entry for each row starting at zero, and for each row it will have a series of 'name=value' pairs, one for each column which was specified in the `SELECT` statement.

```
      while ($row = mysql_fetch_assoc($result)) {
         $this->data_array[] = $row;
      } // while
```

Finally we release the database resource and return the multi-dimensional array containing all the data.

```
      mysql_free_result($result);

      return $this->data_array;

   } // getData
```

I should point out here that this is a simplified version of the code which I actually use in my application. My query string is constructed from several component parts as shown in the following:

```
$query = "SELECT $select_str FROM $from_str $where_str $group_str $having_str $sort_str $limit_str";
```

Each of these component parts can be tailored by instructions from the calling script in order to provide the maximum amount of flexibility. In this way I think I have succeeded in building a single function that can handle a multitude of possibilities.

'insertRecord' Method

When the details of a new database record are input through the client's browser they are received by your PHP script in the $_POST array. It therefore seems logical to me to use the $_POST array as the input to my next function. As usual we start by defining the function name and its argument(s). We also initialise the array of potential error messages.

```
   function insertRecord ($fieldarray)
   {
      $this->errors = array();
```

We then connect to the database using the code described previously:

```
      global $dbconnect, $query;
      $dbconnect = db_connect($this->dbname) or trigger_error("SQL", E_USER_ERROR);
```

Now, using the contents of `$fieldlist` which was set in the class constructor we can edit the input array to filter out any items which do not belong in this database table. This removes the SUBMIT button, for example.

```
      $fieldlist = $this->fieldlist;
      foreach ($fieldarray as $field => $fieldvalue) {
         if (!in_array($field, $fieldlist)) {
            unset ($fieldarray[$field]);
         } // if
      } // foreach
```

We can now construct the query string to insert a new record into the database:

```
      $query = "INSERT INTO $this->tablename SET ";
      foreach ($fieldarray as $item => $value) {
         $query .= "$item='$value', ";
      } // foreach
```

You may have noticed that each 'name=value' pair was appended to the query string with a trailing comma as a separator, so we must remove the final comma like so:

```
      $query = rtrim($query, ', ');
```

Now we can execute the query. Notice here that instead of the default error checking I look specifically for a 'duplicate key' error and return a simple error message rather terminating the whole script with a fatal error.

```
      $result = @mysql_query($query, $dbconnect);
      if (mysql_errno() <> 0) {
         if (mysql_errno() == 1062) {
            $this->errors[] = "A record already exists with this ID.";
         } else {
            trigger_error("SQL", E_USER_ERROR);
         } // if
      } // if
```

The last act is to return control to the calling script.

```
      return;

   } // insertRecord
```

'updateRecord' Method

This routine will update a single record using data which is passed in as an associative array. As with the insertRecord routine this may come directly from the $_POST array. As usual we start by defining the function name and its argument(s). We also initialise the array of potential error messages.

```
   function updateRecord ($fieldarray)
   {
      $this->errors = array();
```

We then connect to the database using the code described previously:

```
      global $dbconnect, $query;
      $dbconnect = db_connect($this->dbname) or trigger_error("SQL", E_USER_ERROR);
```

We then edit the input array to remove any item which does not belong in this database table:

```
      $fieldlist = $this->fieldlist;
      foreach ($fieldarray as $field => $fieldvalue) {
         if (!in_array($field, $fieldlist)) {
            unset ($fieldarray[$field]);
         } // if
      } // foreach
```

In order to update a single record we need to extract the primary key to build a WHERE clause for our database query. At the same time we can also build our UPDATE clause. This can be done within a single loop. Notice that we are using the contents of the class variable `$fieldlist` to identify the primary key for the current table:

```
      $where  = NULL;
      $update = NULL;
      foreach ($fieldarray as $item => $value) {
         if (isset($fieldlist[$item]['pkey'])) {
            $where .= "$item='$value' AND ";
         } else {
            $update .= "$item='$value', ";
```

```
         } // if
      } // foreach
```

Each 'name=value' pair was inserted with a trailing separator which must be removed from the last entry:

```
      $where  = rtrim($where, ' AND ');
      $update = rtrim($update, ', ');
```

Finally we can execute the query and return to the calling script.

```
      $query = "UPDATE $this->tablename SET $update WHERE $where";
      $result = mysql_query($query, $dbconnect) or trigger_error("SQL", E_USER_ERROR);

      return;

   } // updateRecord
```

Notice that by default it is not possible to change the primary key. Although some databases do allow it, most do not, and I have always designed my databases and associated applications accordingly.

'deleteRecord' Method

This routine will delete a single record using data which is passed in as an associative array. As a minimum this array must contain details of the record's primary key. As usual we start by defining the function name and its argument(s). We also initialise the array of potential error messages.

```
   function deleteRecord ($fieldarray)
   {
      $this->errors = array();
```

We then connect to the database using the code described previously:

```
      global $dbconnect, $query;
      $dbconnect = db_connect($this->dbname) or trigger_error("SQL", E_USER_ERROR);
```

We now use the contents of the class variable $fieldlist to identify the primary key for the current table so that we can construct the WHERE clause for our database query:

```
      $fieldlist = $this->fieldlist;
      $where  = NULL;
      foreach ($fieldarray as $item => $value) {
         if (isset($fieldlist[$item]['pkey'])) {
            $where .= "$item='$value' AND ";
         } // if
      } // foreach
```

Each 'name=value' pair was inserted with a trailing separator which must be removed from the last entry:

```
      $where  = rtrim($where, ' AND ');
```

Finally we can execute the query and return to the calling script.

```
      $query = "DELETE FROM $this->tablename WHERE $where";
      $result = mysql_query($query, $dbconnect) or trigger_error("SQL", E_USER_ERROR);

      return;

   } // deleteRecord
} // end class
```

Using this Class

So much for defining the class with its properties and methods, but how do you go about using it in your PHP scripts?

The first step is to create a subclass for each physical database table which extends this base class. This must contain its own class constructor specifically tailored to reflect the details of the database table in question. This is done using code similar to the following:

```
require_once 'default_table.class.inc';
class Sample extends Default_Table
{
    // additional class variables go here
```

```
    function Sample ()
    {
        $this->tablename       = 'sample';
        $this->dbname          = 'foobar';
        $this->rows_per_page   = 15;
        $this->fieldlist       = array('column1', 'column2', 'column3', ...);
        $this->fieldlist['column1'] = array('pkey' => 'y');
        et cetera ...

    } // end class constructor

} // end class
```

Controller scripts

Having created a subclass you are then able to include the class definition in any script and create one or more objects from this class. You are then able to start using the class to communicate with your database, for which you will need a separate script as shown in the following code snippet:

```
include 'sample.class.inc';
$dbobject = new Sample;

// if $where is null then all rows will be retrieved
$where = "column='value'";
// user may specify a particular page to be displayed
if (isset($_GET['pageno'])) {
   $dbobject->setPageno($_GET['pageno']);
} // if
$data = $dbobject->getData($where);
$errors = $dbobject->getErrors();
if (!empty($errors)) {
   // deal with error message(s)
} // if
```

Such scripts are often referred to as "controllers" as they control which methods are called on which objects, when, and in what sequence. It may also be because they represent the "controller" component in the Model-View-Controller design pattern.

All data retrieved will now be available as a multi-dimensional array in `$data` which can be accessed as follows:

```
foreach ($data as $row) {
    foreach ($row as $field => $value) {
        ....
    } // foreach
} // foreach
```

The following values may also be retrieved if required:

- `$dbobject->numrows` will return the total number of rows which satisfied the selection criteria.
- `$dbobject->pageno` will return the current page number based on `$rows_per_page`.
- `$dbobject->lastpage` will return the last page number based on `$rows_per_page`.

In the following code snippets `$fieldarray` may be the `$_POST` array, or it may be constructed within your PHP script.

```
$fieldarray = $dbobject->insertRecord($fieldarray);
$errors = $dbobject->getErrors();

$fieldarray = $dbobject->updateRecord($fieldarray);
$errors = $dbobject->getErrors();

$fieldarray = $dbobject->deleteRecord($fieldarray);
$errors = $dbobject->getErrors();
```

Standard functions

These are some standard functions which I use throughout my software and which can be tailored for use in any application.

db_connect

This is the contents of my 'db.inc' file which I include in every script. As well as opening a connection to your MySQL server it will select the desired database.

```php
$dbconnect  = NULL;
$dbhost     = "localhost";
$dbusername = "****";
$dbuserpass = "****";

$query      = NULL;

function db_connect($dbname)
{
   global $dbconnect, $dbhost, $dbusername, $dbuserpass;

   if (!$dbconnect) $dbconnect = mysql_connect($dbhost, $dbusername, $dbuserpass);
   if (!$dbconnect) {
      return 0;
   } elseif (!mysql_select_db($dbname)) {
      return 0;
   } else {
      return $dbconnect;
   } // if

} // db_connect
```

Error Handler

This is the contents of my 'error.inc' file which I include in every script. It contains my universal error handler which traps every error, and for fatal errors it will display all relevant details on the screen and stop the system. In the event of a database error it will display the contents of the last `$query` string.

```php
set_error_handler('errorHandler');

function errorHandler ($errno, $errstr, $errfile, $errline, $errcontext)
// If the error condition is E_USER_ERROR or above then abort
{
   switch ($errno)
   {
      case E_USER_WARNING:
      case E_USER_NOTICE:
      case E_WARNING:
      case E_NOTICE:
      case E_CORE_WARNING:
      case E_COMPILE_WARNING:
         break;
      case E_USER_ERROR:
      case E_ERROR:
      case E_PARSE:
      case E_CORE_ERROR:
      case E_COMPILE_ERROR:

         global $query;

         session_start();

         if (eregi('^(sql)$', $errstr)) {
            $MYSQL_ERRNO = mysql_errno();
            $MYSQL_ERROR = mysql_error();
            $errstr = "MySQL error: $MYSQL_ERRNO : $MYSQL_ERROR";
         } else {
            $query = NULL;
         } // if

         echo "<h2>This system is temporarily unavailable</h2>\n";
         echo "<b><font color='red'>\n";
         echo "<p>Fatal Error: $errstr (# $errno).</p>\n";
         if ($query) echo "<p>SQL query: $query</p>\n";
         echo "<p>Error in line $errline of file '$errfile'.</p>\n";
         echo "<p>Script: '{$_SERVER['PHP_SELF']}'.</p>\n";
         echo "</b></font>";

         // Stop the system
         session_unset();
         session_destroy();
         die();
      default:
         break;
```

```
   } // switch
} // errorHandler
```

Summary

I hope this tutorial has demonstrated to PHP programmers who are new to Object Oriented programming that it need not be too complicated to implement. What I have demonstrated here uses just some of the basic features of OO programming within PHP, but the results are quite beneficial.

The code I have shown here is just the first step in providing a standard database-access class which can deal with most situations you will encounter. The code in this standard class can then be inherited and reused in any subclass, and where necessary extended on a per-table basis to deal with specific situations.

The more observant of you may have noticed that none of the code I have shown here which updates the database contains any sort of validation. In a follow-up article I will show you how it is possible to enhance this code to provide the following:

- A standard method of initial validation on all user input covering required fields, date fields, numeric fields, et cetera.
- A standard method of validating changes to candidate keys.
- A standard method of dealing with relationships when records are deleted.

I have created a sample application which is based on the code described in this article. You can run this application from my website or download all the source code and run it on your own PC. Please refer to A Sample PHP Application for details.