69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Lecture Sheet on "+MySQL"
Day-22: (ZCE Guidelines)

Working with JavaScript **Time: 2** hours

This lecture provides the learner with information about implementing JavaScript for handling forms with effective form validation

Objectives	Topics
❖ Clear Concept on PHP Coding Methodology	❖ Recap of All Concepts in Veiw of ZCE

echo isset ($x);

A call to isset() will return true if a variable exists and has a value other than NULL.

Comparing Arrays

Array-to-array comparison is a relatively rare occurrence, but it can be performed using another set of operators. Like for other types, the equivalence and identity operators can be used for this purpose:

```
$a = array (1, 2, 3);

$b = array (1 => 2, 2 => 3, 0 => 1);

$c = array ('a' => 1, 'b' => 2, 'c' => 3);

var_dump ($a == $b); // True

var_dump ($a === $b); // False

var_dump ($a == $c); // True

var_dump ($a === $c); // False
```

As you can see, the equivalence operator == returns true if both arrays have the same number of elements with the same values and keys, regardless of their order. The identity operator ===, on the other hand, returns true only if the array contains the same key/value pairs in the same order. Similarly, the inequality and non-identity operators can determine whether two arrays are different:

```
$a = array (1, 2, 3);

$b = array (1 => 2, 2 => 3, 0 => 1);
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
var_dump ($a != $b); // False

var_dump ($a !== $b); // True
```

Once again, the inequality operator only ensures that both arrays contain the same elements with the same keys, whereas the non-identity operator also verifies their position.

Counting, Searching and Deleting Elements

The size of an array can be retrieved by calling the count() function:

```
$a = array (1, 2, 4);

$b = array();

$c = 10;

echo count ($a); // Outputs 3

echo count ($b); // Outputs 0

echo count ($c); // Outputs 1
```

As you can see, count() cannot be used to determine whether a variable contains an array—since running it on a scalar value will return one. The right way to tell whether a variable contains an array is to use is_array() instead.

A similar problem exists with determining whether an element with the given key exists. This is often done by calling isset():

```
$a = array ('a' => 1, 'b' => 2);
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
echo isset ($a['a']); // True

echo isset ($a['c']); // False
```

However, isset() has the major drawback of considering an element whose value is NULL—which is perfectly valid—as inexistent:

```
$a = array ('a' => NULL, 'b' => 2);

echo isset ($a['a']); // False
```

The correct way to determine whether an array element exists is to use array_key_exists() instead:

```
$a = array ('a' => NULL, 'b' => 2);

echo array_key_exists ($a['a']); // True
```

Obviously, neither these functions will allow you to determine whether an element with a given value exists in an array—this is, instead, performed by the in_array() function:

```
$a = array ('a' => NULL, 'b' => 2);

echo in_array ($a, 2); // True
```

Finally, an element can be deleted from an array by unsetting it:

```
$a = array ('a' => NULL, 'b' => 2);

unset ($a['b']);

echo in_array ($a, 2); // False
```

Flipping and Reversing

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

There are two functions that have rather confusing names and that are sometimes misused: array_flip() and array_reverse(). The first of these two functions inverts the value of each element of an array with its key:

```
$a = array (’a’, ’b’, ’c’);

var_dump (array_flip ($a));
```

This outputs:

```
array(3) {

   ["a"]=>
  int(0)

   ["b"]=>
  int(1)

   ["c"]=>
  int(2)
}
```

On the other hand, array_reverse() actually inverts the order of the array’s elements, so that the last one appears first:

```
$a = array (’x’ => ’a’, 10 => ’b’, ’c’);

var_dump (array_reverse ($a));
```

Note how key key association is only lost for those elements whose keys are numeric:

```
array(3) {

   [0]=>
  string(1) "c"

   [1]=>
 string(1) "b"
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
    ["x"]=>
  string(1) "a"
}
```

The Array Pointer

Each array has a pointer that indicates the "current" element of an array in an it eration. The pointer is used by a number of different constructs, but can only be manipulated through a set of functions and does not affect your ability to access individual elements of an array, nor is it affected by most "normal" array operations.

The pointer is, in fact, a handy way of maintaining the iterative state of an array without needing an external variable to do the job for us.

The most direct way of manipulating the pointer of an array is by using a series of functions designed specifically for this purpose. Upon starting an iteration over an array, the first step is usually to reset the pointer to its initial position using the reset() function; after that, we can move forward or backwards by one position by using prev() and next() respectively. At any given point, we can access the value of the current element using current() and its key using key(). Here's an example:

```
$array = array('foo' => 'bar', 'baz', 'bat' => 2);
function displayArray($array) {
    reset($array);
    while (key($array) !== null) {
        echo key($array) .": " .current($array) . PHP_EOL;
        next($array);
    }
}
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Here, we have created a function that will display all the values in an array. First, we call reset() to rewind the internal array pointer. Next, using a while loop, we display the current key and value, using the key() and current() functions. Finally, we advance the array pointer, using next(). The loop continues until we no longer have a valid key.

Passive Iteration

The array_walk() function and its recursive cousin array_walk_recursive() can be used to perform an iteration of an array in which a user-defined function is called. Here's an example:

```
    function setCase(&$value, &$key)
{
  $value = strtoupper($value);
  }

    $type = array('internal', 'custom');

    $output_formats[] = array('rss', 'html', 'xml');

    $output_formats[] = array('csv', 'json');

    $map = array_combine($type, $output_formats);

    array_walk_recursive($map, 'setCase');

    var_dump($map);
```

Using the custom setCase() function, a simple wrapper for strtoupper(), we are able to convert each each of the array's values to uppercase. One thing to note about **array_walk_recursive()** is that it will not call the user-defined function on anything but scalar values; because of this, the first set of keys, internal and custom, are never

passed in.

The resulting array looks like this:

```
array(2) {
  ["internal"]=>
  &array(3) {
    [0]=>
    string(3) "RSS"
    [1]=>
    string(4) "HTML"
    [2]=>
    string(3) "XML"
  }
  ["custom"]=>
  &array(2) {
    [0]=>
    string(3) "CSV"
    [1]=>
    string(4) "JSON"
  }
}
```

Sorting Arrays

There are a total of eleven functions in the PHP core whose only goal is to provide various methods of sorting the contents of an array. The simplest of these is sort(), which sorts an array based on its values:

```
$array = array('a' => 'foo', 'b' => 'bar', 'c' => 'baz');
sort($array);
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
var_dump($array);
```

As you can see, sort() modifies the actual array it is provided, since the latter is

passed by reference. This means that you cannot call this function by passing anything other than a single variable to it.

The result looks like this:

```
array(3) {
   [0]=>
 string(3) "bar"
   [1]=>
 string(3) "baz"
   [2]=>
 string(3) "foo"
}
```

Thus, sort() effectively destroys all the keys in the array and renumbers its elements

starting from zero. If you wish to maintain key association, you can use asort() instead:

```
$array = array(’a’ => ’foo’, ’b’ => ’bar’, ’c’ => ’baz’);
asort($array);
var_dump($array);
```

This code will output something similar to the following:

```
array(3) {
        ["b"]=>
string(3) "bar"
        ["c"]=>
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
string(3) "baz"

        ["a"]=>     string(3) "foo"

    }
```

Both sort() and asort() accept a second, optional parameter that allows you to specify how the sort operation takes place:

SORT_REGULAR Compare items as they appear in the array, without performing any kind of conversion. This is the default behaviour.

SORT_NUMERIC Convert each element to a numeric value for sorting purposes.

SORT_STRING Compare all elements as strings.

Both sort() and asort() sort values in ascending order. To sort them in descending order, you can use rsort() and arsort().

The sorting operation performed by sort() and asort() simply takes into consideration either the numeric value of each element, or performs a byte-by-byte comparison of strings values. This can result in an "unnatural" sorting order—for example, the string value '10t' will be considered "lower" than '2t' because it starts with the character 1, which has a lower value than 2. If this sorting algorithm doesn't work well for your needs, you can try using natsort() instead:

```
$array = array('10t', '2t', '3t');

natsort($array);

var_dump($array);
```

This will output:

```
array(3) {

    [1]=>
```

```
 string(2) "2t"

   [2]=>
 string(2) "3t"

   [0]=>
 string(3) "10t"

}
```

The natsort() function will, unlike sort(), maintain all the key-value associations in the array. A case-insensitive version of the function, natcasesort() also exists, but there is no reverse-sorting equivalent of rsort().

Other Sorting Options

In addition to the sorting functions we have seen this far, PHP allows you to sort by key (rather than by value) using the ksort() and krsort() functions, which work analogously to sort() and rsort():

```
$a = array ('a' => 30, 'b' => 10, 'c' => 22);

ksort($a);

var_dump ($a);
```

This will output:

```
array(3) {

   ["a"]=>
 int(30)

   ["b"]=>        int(10)

   ["c"]=>
 int(22)

}
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Finally, you can also sort an array by providing a user-defined function:

```
function myCmp ($left, $right)
{
    // Sort according to the length of the value.
    // If the length is the same, sort normally
    $diff = strlen ($left) - strlen ($right);
   if (!$diff) {
        return strcmp ($left, $right);
   }
   return $diff;
}
$a = array (
    fthree',
    f2two',
    fone',
    ftwo'
);
usort ($a, 'myCmp');
 var_dump ($a);
```

This short script allows us to sort an array by a rather complicated set of rules: first, we sort according to the length of each element's string representation. Elements whose values have the same length are further sorted using regular string

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

comparison rules; our user-defined function must return a value of zero if the two values

are to be considered equal, a value less than zero if the left-hand value is lower than the right-hand one, and a positive number otherwise. Thus, our script produces this output:

```
array(4) {
  [0]=>
 string(3)  "one"
  [1]=>
 string(3)  "two"
  [2]=>
 string(4)  "2two"
  [3]=>
 string(5)  "three"
}
```

As you can see, usort() has lost all key-value associations and renumbered our array; this can be avoided by using uasort() instead. You can even sort by key (instead of by value) by using uksort(). Note that there is no reverse-sorting version of any of these functions—because reverse sorting can performed by simply inverting the comparison rules of the user-defined function:

```
function myCmp ($left, $right)
{
   // Reverse-sort according to the length of the value.
   // If the length is the same, sort normally
   $diff = strlen ($right) - strlen ($left);
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
        if (!$diff) {
         return strcmp ($right, $left);

        }

        return $diff;
    }
```

This will result in the following output:

```
    array(4) {

      [0]=>
     string(5)    "three"

      [1]=>
     string(4)    "2two"

      [2]=>
     string(3)    "two"

      [3]=>
     string(3)    "one"

    }
```

```
$cards = array (1, 2, 3, 4);


shuffle ($cards);

var_dump ($cards);
```

Arrays as Stacks, Queues and Sets

Arrays are often used as stacks (Last In, First Out, or LIFO) and queue (First In, First Out, or FIFO) structures. PHP simplifies this approach by providing a set of functions
can be used to push and pull (for stacks) and shift and unshift (for queues) elements from an array.

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

We'll take a look at stacks first:

```
$stack = array();

array_push($stack, 'bar', 'baz');

var_dump($stack);

    $last_in = array_pop($stack);

    var_dump($last_in, $stack);
```

In this example, we first, create an array, and we then add two elements to it using

array_push(). Next, using array_pop(), we extract the last element added to the array,

resulting in this output:

```
array(2) {

   [0]=>
string(3) "bar"

   [1]=>
string(3) "baz"

}

string(3) "baz"

array(1) {

   [0]=>
string(3) "bar"

}
```

As you have probably noticed, when only one value if being pushed, array_push() is

equivalent to adding an element to an array using the syntax $a[] = $value. In fact,

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

the latter is much faster, since no function call takes place and, therefore, should be the preferred approach unless you need to add more than one value.

If you intend to use an array as a queue, you can add elements to the beginning and extract them from the end by using the array_unshift() and array_shift() functions:

```
$stack = array('qux', 'bar', 'baz');
$first_element = array_shift($stack);
  var_dump($stack);
  array_unshift($stack, 'foo');
var_dump($stack);
```

In this example, we use array_shift() to push the first element out of the array. Next, using array_unshift(), we do the reverse and add a value to the beginning of the array. This example results in:

```
array(2) {
  [0]=>
 string(3) "bar"
  [1]=>
 string(3) "baz"
}
array(3) {
  [0]=>
 string(3) "foo"
  [1]=>
 string(3) "bar"
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
  [2]=>
 string(3) "baz"

}
```

Set Functionality

Some PHP functions are designed to perform set operations on arrays. For example, array_diff() is used to compute the difference between two arrays:

```
$a = array (1, 2, 3);

$b = array (1, 3, 4);

var_dump (array_diff ($a, $b));
```

The call to array_diff() in the code above will cause all the values of $a that do not also appear in $b to be retained, while everything else is discarded:

```
array(1) {
 [1]=>
int(2)
}
```

Note that the keys are ignored—if you want the difference to be computed based on key-value pairs, you will have to use array_diff_assoc() instead, whereas if you want it to be computed on keys alone, array_diff_key() will do the trick. Both of these functions have user-defined callback versions called array_diff_uassoc() and array_diff_ukey() respectively.

Conversely to array_diff(), array_intersect() will compute the intersection between two arrays:

```
$a = array (1, 2, 3);
```

```
$b = array (1, 3, 4);

var_dump (array_intersect ($a, $b));
```

In this case, only the values that are included in both arrays are retained in the result:

```
array(2) {

   [0]=>
 int(1)

   [2]=>
 int(3)

}
```

Like with array_diff(), array_intersect only keeps in consideration the value of

each element; PHP provides array_intersect_key() and array_intersect_assoc()

versions for key- and key/value-based intersection, together with their callback variants array_intersect_ukey() and array_intersect_uassoc().

HereDoc

Heredoc strings can be used in almost all situations in which a string is an appropriate value. The only exception is the declaration of a class property (explained in

the Object Oriented Programming With PHP chapter), where their use will result in a

parser error:

```
class Hello {

   public $greeting = <<<EOT

Hello World

EOT;

}
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Generic Formatting

If you are not handling numbers or currency values, you can use the printf() family of functions to perform arbitrary formatting of a value. All the functions in this group perform in an essentially identical way: they take an input string that specifies the output format and one or more values. The only difference is in the way they return their results: the "plain" printf() function simply writes it to the script's output, while other variants may return it (sprintf()), write it out to a file (fprintf()), and so on.

The formatting string usually contains a combination of literal text—that is copied directly into the function's output—and specifiers that determine how the input should be formatted. The specifiers are then used to format each input parameter in the order in which they are passed to the function (thus, the first specifier is used to format the first data parameter, the second specified is used to format the second parameter, and so on).

A formatting specifier always starts with a percent symbol (if you want to insert a literal percent character in your output, you need to escape it as %%) and is followed by a type specification token, which identifies the type of formatting to be applied; a number of optional modifiers can be inserted between the two to affect the output:

- A sign specifier (a plus of minus symbol) to determine how signed numbers are to be rendered
- A padding specifier that indicates what character should be used to make up the required output length, should the input not be long enough on its own

• An alignment specifier that indicates if the output should be left or right aligned

• A numeric width specifier that indicates the minimum length of the output

• A precision specifier that indicates how many decimal digits should be displayed for floating-point numbers

Cookies

Cookies allow your applications to store a small amount of textual data (typically, 4-6kB) on a Web client. There are a number of possible uses for cookies, although their most common one is maintaining session state (explained in the next section).

Cookies are typically set by the server using a response header, and subsequently made available by the client as a request header.

You should not think of cookies as a secure storage mechanism. Although you can transmit a cookie so that it is exchanged only when an HTTP transaction takes place securely (e.g.: under HTTPS), you have no control over what happens to the cookie data while it's sitting at the client's side—or even whether the client will accept your cookie at all (most browsers allow their users to disable cookies). Therefore, cookies should always be treated as "tainted" until proven otherwise—a concept that we'll examine in the Security chapter.

To set a cookie on the client, you can use the setcookie() function:

```
setcookie("hide_menu", "1");
```

This simple function call sets a cookie called “hide_menu” to a value of 1 for the remainder of the users browser session, at which time it is automatically deleted.

Should you wish to make a cookie persist between browser sessions, you will need to provide an expiration date. Expiration dates are provided to setcookie() in the UNIX timestamp format (the number of seconds that have passed since January 1, 1970). Remember that a user or their browser settings can remove a cookie at any time—therefore, it is unwise to rely on expiration dates too much.

```
setcookie("hide_menu", "1", time() + 86400);
//This will instruct the browser to (try to) hang on to the cookie for a day.
```

There are three more arguments you can pass to setcookie(). They are, in order:

- path—allows you to specify a path (relative to your website’s root) where the cookie will be accessible; the browser will only send a cookie to pages within this path.
- domain—allows you to limit access to the cookie to pages within a specific domain or hostname; note that you cannot set this value to a domain other than the one of the page setting the cookie (e.g.: the host www.phparch.com can set a cookie for hades.phparch.com, but not for www.microsoft.com).

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

• secure—this requests that the browser only send this cookie as part of its re-

quest headers when communicating under HTTPS.

Accessing Cookie Data

Cookie data is usually sent to the server using a single request header. The PHP interpreter takes care of automatically separating the individual cookies from the header

and places them in the $_COOKIE superglobal array:

```
if ($_COOKIE['hide_menu'] == 1) {
    // hide menu
}
```

Cookie values must be scalar; of course, you can create arrays using the same array

notation that we used for $_GET and $_POST:

```
setcookie("test_cookie[0]", "foo");

setcookie("test_cookie[1]", "bar");

setcookie("test_cookie[2]", "bar");
```

At the next request, $_COOKIE['test_cookie'] will automatically contain an array. You should, however, keep in mind that the amount of storage available is severely limited—therefore, you should keep the amount of data you store in cookies to a minimum, and use sessions instead.

Remember that setting cookies is a two-stage process: first, you send the cookie to the

client, which will then send it back to you at the next request. Therefore, the $_COOKIE

array will not be populated with new information until the next request comes along.

There is no way to "delete" a cookie—primarily because you really have no control over how cookies are stored and managed on the client side. You can, however, call setcookie with an empty string, which will effectively reset the cookie:

```
setcookie("hide_menu", false, -3600);
```

Sessions

HTTP is a stateless protocol; this means that the webserver does not know (or care) whether two requests comes from the same user; each request is instead handled without regard to the context in which it happens. Sessions are used to create a measure of state in between requests—even when they occur at large time intervals from each other.

Sessions are maintained by passing a unique session identifier between requests—typically in a cookie, although it can also be passed in forms and GET query arguments. PHP handles sessions transparently through a combination of cookies and URL rewriting, when **session.use_trans_sid** is turned on in **php.ini** (it is off by default in PHP5) by generating a unique session ID and using it track a local data store (by default, a file in the system's temporary directory) where session data is saved at the end of every request.

Sessions are started in one of two ways. You can either set PHP to start

a new session automatically whenever a request is received by changing the

session.auto_start configuration setting in your php.ini file, or explicitly call

session_start() at the beginning of each script. Both approaches have their advantages and drawbacks. In particular, when sessions are started automatically, you

obviously do not have to include a call to session_start() in every script. However,

the session is started before your scripts are executed; this denies you the opportunity to load your classes before your session data is retrieved, and makes storing

objects in the session impossible.

In addition, session_start() must be called before any output is sent to the

browser, because it will try to set a cookie by sending a response header.

In the interest of security, it is a good idea to follow your call to session_start() with a

call to **session_regenerate_id()** whenever you change a user's privileges to prevent

"session fixation" attacks. We explain this problem in greater detail in the Security

chapter.

Accessing Session Data

Once the session has been started, you can access its data in the $_SESSION super global array:

```
// Set a session variable

$_SESSION['hide_menu'] = true;

// From here on, we can access hide_menu in $_SESSION
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
if ($_SESSION['hide_menu']) {
  // Hide menu
}
```

Transactions

Many database engines implement transaction blocks, which are groups of operations that are committed (or discarded) atomically, so that either all of them are applied to the database, or none.

A **transaction** starts with the issuing of a START TRANSACTION statement. From here on, all further operations take place in a "sandbox" that does not affect any other user or, indeed, the database itself until the transaction is either complete using the COMMIT statement, or "undone" using ROLLBACK. In the latter case, all the operations that took place since the transaction started are simply discarded and do not affect the database at all.

Here are two examples:

```
START TRANSACTION
DELETE FROM book WHERE isbn LIKE '0655%'
UPDATE book_chapter set chapter_number = chapter_number + 1
ROLLBACK
START TRANSACTION
UPDATE book SET id = id + 1
DELETE FROM book_chapter WHERE isbn LIKE '0433%'
COMMIT
```

The first transaction block will essentially cause no changes to the database, since it ends with a rollback statement. Keep in mind that this condition usually takes place in scenarios in which multiple operations are interdependent and must all succeed in order for the transaction to be completed—typically, this concept is illustrated with the transfer of money from one bank account to another: the "transaction," in this case, isn't complete until the money has been taken from the source account and deposited in the destination account. If, for any reason, the second part of the operation isn't possible, the transaction is rolled back so that the money doesn't just "disappear."

Prepared Statements

For the most part, the only thing that changes in your application's use of SQL is the data in the queries you pass along to the database system; the queries themselves almost never do. This means—at the very least—that your database system has to parse and compile the SQL code that you pass along every time. While this is not a large amount of overhead, it does add up—not to mention the fact that you do need to ensure that you escape all of your data properly every time.

Many modern database systems allow you to short-circuit this process by means of a technique known as a prepared statement. A prepared statement is, essentially, the template of an SQL statement that has been pre-parsed and compiled and is ready to be executed by passing it the appropriate data. Each database system implements this in a different way, but, generally speaking, the process works in three steps: first, you create the prepared statement, replacing your data with a set of "markers"—such

as question marks, or named entities. Next, you load the data in the statement, and finally execute it. Because of this process, you do not have to mix data and SQL code in the same string—which clearly reduces the opportunity for improper escaping and, therefore, for security issues caused by malicious data.

Design Pattern Theory

As we mentioned in the previous section, design patterns are nothing more than streamlined solutions to common problems. In fact, design patterns are not really about code at all—they simply provide guidelines that you, the developer, can translate into code for pretty much every possible language. In this chapter, we will provide a basic description of some of the simpler design patterns, but, as the exam concerns itself primarily with the theory behind them, we will, for the most part, stick to explaining how they work in principle.

Even though they can be implemented using nothing more than procedural code, design patterns are best illustrated using OOP. That's why it's only with PHP 5 that they have really become relevant to the PHP world: with a proper object-oriented architecture in place, building design patterns is easy and provides a tried-and-true method for developing robust code.

The Singleton Pattern

The Singleton is, probably, the simplest design pattern. Its goal is to provide access to a single resource that is never duplicated, but that is made available to any portion of an application that requests it without the need to keep track of its existence. The most typical example of this pattern is a database connection, which normally only needs to be created once at the beginning of a script and then used throughout its code. Here's an example implementation:

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
class DB {

    private static $_singleton;

    private $_connection;

    private function __construct()

    {

       $this->_connection = mysql_connect();

    }

    public static function getInstance()

    {

if (is_null (self::$_singleton)) {
self::$_singleton = new DB();}

        return self::$_singleton;

    }

}

$db = DB::getInstance();
```

Our implementation of the DB class takes advantage of a few advanced OOP concepts

that are available in PHP 5: we have made the constructor private, which effectively

ensures that the class can only be instantiated from within itself. This is, in fact, done

in the getInstance() method, which checks whether the static property $_connection

has been initialized and, if it hasn't, sets it to a new instance of DB. From this point on,

getInstance() will never attempt to create a new instance of DB, and instead always

return the initialized $_connection, thus ensuring that a database connection is not

created more than once.

The Factory Pattern

The Factory pattern is used in scenarios where you have a generic class (the factory) that provides the facilities for creating instances of one or more separate "specialized" classes that handle the same task in different ways.

A good situation in which the Factory pattern provides an excellent solution is the management of multiple storage mechanisms for a given task. For example, consider configuration storage, which could be provided by data stores like INI files, databases or XML files interchangeably. The API that each of these classes provides is the same (ideally, implemented using an interface), but the underlying implementation changes. The Factory pattern provides us with an easy way to return a different data store class depending on either the user's preference, or a set of environmental factors:

```
class Configuration {
    const STORE_INI = 1;
    const STORE_DB = 2;
    const STORE_XML = 3;
    public static function getStore($type = self::STORE_XML)
    {
        switch ($type) {
            case self::STORE_INI:
return new Configuration_Ini();
            case self::STORE_DB:
```

```
    return new Configuration_DB();

                case self::STORE_XML:
    return new Configuration_XML();

                default:
                  throw new Exception("Unknown Datastore Specified.");

              }

           }

        }

    class Configuration_Ini {
        // ...
     }

    class Configuration_DB {
        // ...
    }

    class Configuration_XML {
       // ...
    }

    $config = Configuration::getStore(Configuration::STORE_XML);
```

The Registry Pattern

By taking the Singleton pattern a little further, we can implement the Registry pattern. This allows us to use any object as a Singleton without it being written specifically that way.

The Registry pattern can be useful, for example, if, for the bulk of your application,

you use the same database connection, but need to connect to an alternate database

to perform a small set of tasks every now and then. If your DB class is implemented

as a Singleton, this is impossible (unless you implement two separate classes, that

is)—but a Registry makes it very easy:

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
class Registry {

    private static $_register;

    public static function add(&$item, $name = null)

    {

        if (is_object($item) && is_null($name)) {
$name = get_class($item);

        } elseif (is_null($name)) {

            $msg = "You must provide a name for non-objects";

            throw new Exception($msg);

        }

        $name = strtolower($name);

        self::$_register[$name] = $item;

    }

    public static function &get($name)

    {

        $name = strtolower($name);

        if (array_key_exists($name, self::$_register)) {

            return self::$_register[$name];

        } else {

            $msg = "'$name' is not registered.";

            throw new Exception($msg);

        }

    }

    public static function exists($name)
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
    {
        $name = strtolower($name);
        if (array_key_exists($name, self::$_register)) {
            return true;
        } else {
return false;
     }
    }
}
$db = new DB();
Registry::add($db);
// Later on
if (Registry::exists('DB')) {
    $db = Registry::get('DB');
} else {
    die('We lost our Database connection somewhere. Bear with us.');
}
```

The Model-View-Controller Pattern

Unlike the patterns we have seen this far, Model-View-Controller (MVC) is actually quite complex. Its goal is that of providing a methodology for separating the business logic (model) from the display logic (view) and the decisional controls (controller).

In a typical MVC setup, the user initiates an action (even a default one) by calling the Controller. This, in turn, interfaces with the Model, causing it to perform some sort of action and, therefore, changing its state. Finally, the View is called, thus causing the user

interface to be refreshed to reflect the changes in the Model and the action requested of the Controller, and the cycle begins anew.

The clear advantage of the MVC pattern is its clear-cut approach to separating each domain of an application into a separate container. This, in turn, makes your applications easier to maintain and to extend, particularly because you can easily modularize each element, minimizing the possibility of code duplication.

The ActiveRecord Pattern

The last pattern that we will examine is the ActiveRecord pattern. This is used to encapsulate access to a data source so that the act of accessing its components—both for reading and for writing—is, in fact, hidden within the class that implements the pattern, allowing its callers to worry about using the data, as opposed to dealing with the database.

The concept behind ActiveRecord is, therefore, quite simple, but its implementation can be very complicated, depending on the level of functionality that a class based on this pattern is to provide. This is usually caused by the fact that, while developers tend to deal with individual database fields individually and interactively, SQL deals with them as part of rows that must be written back to the database atomically. In addition, the synchronization of data within your script to the data inside the database can be very challenging, because the data may change after you've fetched it from the database without giving your code any notice.