HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

BBM 487
SOFTWARE ENGINEERING LAB

CODING STANDARD

Özdeş Öztürk	Ekin Kandemir	Kadir Bulut
21228635	21228434	21228137

Library Book Loan System
Group #7
25/04/2017

Library Book Loan System(LBLS)
Coding Standard

1. Introduction

This document describes rules and recommendations for developing applications and class libraries using the PHP Language. The goal is to define guidelines to enforce consistent style and formatting and help developers avoid common pitfalls and mistakes. The guidelines are similar to Pear standards in many ways, but differ in some key respects. This section of the standard comprises what should be considered the standard coding elements that are required to ensure a high level of technical interoperability between shared PHP code.

1.1. What is PHP

PHP (PHP was originally an acronym for Personal Home Pages, but is now a recursive acronym for PHP: Hypertext Preprocessor) is a widely-used open source general-purpose scripting language that is especially suited for web development and can be embedded into HTML.

An example:

```
<!DOCTYPE HTML>
<html>
   <head>
      <title>Example</title>
   </head>
   <body>

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

   </body>
</html>
```

PHP was originally developed by the Danish Greenlander Rasmus Lerdorf, and was subsequently developed as open source. PHP is not a proper web standard - but an open-source technology. PHP is neither real programming language - but PHP lets you use so-called scripting in your documents.

1.2. Keywords

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

2. Overview

- Files MUST use only <?php and <?= tags.
- Files MUST use only UTF-8 without BOM for PHP code.
- Files SHOULD either declare symbols (classes, functions, constants, etc.) or cause side-effects (e.g. generate output, change .ini settings, etc.) but SHOULD NOT do both.
- Namespaces and classes MUST follow an “autoloading” PSR: [PSR-0, PSR-4].
- Method names MUST be declared in camelCase.

2.1 Single and Double Quotes

Use single and double quotes when appropriate. If you’re not evaluating anything in the string, use single quotes. You should almost never have to escape quotes in a string, because you can just alternate your quoting style, like so:

```
                echo '<tr>
                            <th>Username</th>
                            <th>Password</th>
                            <th>Name</th>
                            <th>Surname</th>
                            <th></th>
                      </tr>
                <tbody>';


<input type="hidden" id="bookID" name="bookID" value="'.$row['bookID'].'">
```

2.2 Indentation

Your indentation should always reflect logical structure. Use real tabs and not spaces, as this allows the most flexibility across clients. Using only spaces, and not mixing spaces with tabs, helps to avoid problems with diffs, patches, history, and annotations. The use of spaces also makes it easy to insert fine-grained sub-indentation for inter-line alignment. Code MUST use an indent of 4 spaces, and MUST NOT use tabs for indenting.

2.3 Lines

There MUST NOT be a hard limit on line length. The soft limit on line length MUST be 120 characters; automated style checkers MUST warn but MUST NOT error at the soft limit.
Blank lines MAY be added to improve readability and to indicate related blocks of code.There MUST NOT be more than one statement per line.
Lines SHOULD NOT be longer than 80 characters; lines longer than that SHOULD be split into multiple subsequent lines of no more than 80 characters each.There MUST NOT be trailing whitespace at the end of non-blank lines.

2.4 Keywords and True/False/Null

PHP keywords MUST be in lower case.
The PHP constants true, false, and null MUST be in lower case.

2.5 Brace Style

Braces shall be used for all blocks in the style shown here:

```
function printInsertUserMessage($mode) {/* prints a message for inserting user */
        $str = "SUCCESS";
        $message = "The user has been inserted successfuly";
        $color = "green";
        if($mode == 0) {
                $message = "User inserting has been failed";
                $color = "red";
                $str = "FAILURE";
        } else if($mode == 2) {
                $message = "The user already exist!!";
                $color = "red";
                $str = "FAILURE";
        }
}
```

Furthermore, if you have a really long block, consider whether it can be broken into two or more shorter blocks or functions. If you consider such a long block unavoidable, please put a short comment at the end so people can tell at glance what that ending brace ends – typically this is appropriate for a logic block, longer than about 70 rows, but any code that's not intuitively obvious can be commented.

Braces should always be used, even when they are not required:

```
<form action="update_user_page.php" method="POST">
<input type="hidden" id="username"
name="username" value="'.$row['username'].'">
<input type="hidden" id="password"
name="password" value="'.$row['password'].'">
<input type="hidden" id="name"
name="name" value="'.$row["name"].'">
<input type="hidden" id="surname"
name="surname" value="'.$row["surname"].'">
<input type="hidden" id="userID"
name="userID" value="'.$row['userID'].'">
<button class="modify_btn" type="submit"
style="letter-spacing: 0px;">UPDATE</button>
</form>
```

2.6 Control Structures

- There MUST be one space after the control structure keyword
- There MUST NOT be a space after the opening parenthesis
- There MUST NOT be a space before the closing parenthesis
- There MUST be one space between the closing parenthesis and the opening brace
- The structure body MUST be indented once.
- The closing brace MUST be on the next line after the body

2.7 Use elseif, Not else if

Else if is not compatible with the colon syntax for if|elseif blocks. For this reason, use elseif for conditionals.

```
else if(($old_a_surname != $a_surname)
&& ($a_surname != NULL)) {
/* author surname is changed */

                        /* get id of new author */
                        $a_id = getAuthorId($old_a_name, $a_surname);
                }
        }
```

2.8 While, do while

A while statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
while($row = mysqli_fetch_array($books)) {          /* for every book */
                        /* show book properties */
                        echo '<tr>
                                        <td>'.$row["title"].'</td>
                                        <td>'.$row["a_name"].' '.$row["a_surname"].'</td>
                                        <td>'.$row["p_name"].'</td>
```

```
                                      <td>'.$row["stock_num"].'</td>
                                      <td> …
```

2.9 No Shorthand PHP Tags

Important: Never use shorthand PHP start tags. Always use full PHP tags.
Correct:

```
<?php ... ?>
<?php echo $var; ?>
```

Incorrect:

```
<? ... ?>
<?= $var ?>
```

2.10 Remove Trailing Spaces

Remove trailing whitespace at the end of each line of code. Omitting the closing PHP tag at the end of a file is preferred. If you use the tag, make sure you remove trailing whitespace.

2.11 Space Usage

Always put spaces after commas, and on both sides of logical, comparison, string and assignment operators.

```
$x = $foo['bar']; // correct
$x = $foo[ 'bar' ]; // incorrect

$x = $foo[0]; // correct
$x = $foo[ 0 ]; // incorrect

$x = $foo[ $bar ]; // correct
$x = $foo[$bar]; // incorrect

if(($old_p_name != $p_name) && ($p_name != NULL))
```

2.12 Formatting SQL Statements

When formatting SQL statements you may break it into several lines and indent if it is sufficiently complex to warrant it. Most statements work well as one line though. Always capitalize the SQL parts of the statement like

```
$sql = mysqli_query($_SESSION["conn"], "select userID
                                         from users
                                         where username = '".$username."'");
```

2.13 Database Queries

Avoid touching the database directly. If there is a defined function that can get the data you need, use it. Database abstraction (using functions instead of queries) helps keep your code forward-compatible and, in cases where results are cached in memory, it can be many times faster.

```
$sql = mysqli_query($_SESSION["conn"], "select userID
                                         from users
                                         where username = '".$username."'");

        if(mysqli_num_rows($sql))  { /* if a record already exist */
```

```
                return -2;
        }
        else {  /* record does not exist */
                /* create new book record */
                $query = "insert into users(username, password, name, surname, authority)
                                                        values('".$username."', '".$pass."', '".$name."',
'".$surname."', ".$authority.")";
                $insert = mysqli_query($_SESSION["conn"], $query);

                if($insert == NULL)  { /* insertion failed */
                        return -1;
                }
                else { /* insertion success */
                        return 1;
                }
        }
```

2.14 Naming Conventions

Use lowercase letters in variable, action, and function names (camelCase). Separate words via underscores. Don't abbreviate variable names unnecessarily; let the code be unambiguous and self-documenting.

function addUser($username, $pass, $name, $surname, $authority) {

Files should be named descriptively using lowercase letters. Hyphens should separate words.

update_user.php

2.15 Self-Explanatory Flag Values for Function Arguments

Since PHP doesn't support named arguments, the values of the flags are meaningless, and each time we come across a function call like the examples above, we have to search for the function definition. The code can be made more readable by using descriptive string values, instead of booleans. When more words are needed to describe the function parameters, an $args array may be a better pattern.

function getAuthorId($a_name, $a_surname) {…}

2.16 Yoda Conditions

```
        if(($title == NULL) || ($a_name == NULL)
                            || ($a_surname == NULL)
                            || ($publisher_name == NULL)) {
                           /* control params - just in case */
                return -1;
        }
        $p_id = -1;
```

When doing logical comparisons involving variables, always put the variable on the right side and put constants, literals, or function calls on the left side. If neither side is a variable, the order is not important.A little bizarre, it is, to read. Get used to it, you will. This applies to ==, !=, ===, and !==. Yoda conditions for <, >, <= or >= are significantly more difficult to read and are best avoided.

2.17 Clever Code

In general, readability is more important than cleverness or brevity.

```
        if(($title == NULL) || ($a_name == NULL)
                            || ($a_surname == NULL)
                            || ($publisher_name == NULL))
```

Although the above line is clever, it takes a while to grok if you're not familiar with it. So, just write it like this:

```
        if ( ! isset( $var ) ) {
  $var = some_function();
}
```

2.18 Error Control Operator

PHP supports one error control operator: the at sign (@). When prepended to an expression in PHP, any error messages that might be generated by that expression will be ignored.

```
$host="localhost";
$db="lbls";
$user="root";
$pass="";
$conn=@mysqli_connect($host,$user,$pass, $db) or die(mysqli_error());
```

Warning: Currently the "@" error-control operator prefix will even disable error reporting for critical errors that will terminate script execution. Among other things, this means that if you use "@" to suppress errors from a certain function and either it isn't available or has been mistyped, the script will die right there with no indication as to why.