Table of Contents

PHP Coding Style

PHP File Formatting (http://framework.zend.com/manual/1.10/en/coding-standard.php-file-formatting.html)

General

For files that contain only PHP code, the closing tag ("?>") is never permitted. It is not required by PHP, and omitting it´ prevents the accidental injection of trailing white space into the response.

Indentation

Indentation should consist of 4 spaces. Tabs are not allowed.

Maximum Line Length

The target line length is 80 characters. That is to say developers should strive keep each line of their code under 80 characters where possible and practical. However, longer lines are acceptable in some circumstances. The maximum length of any line of PHP code is 120 characters.

Line Termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character. Linefeed characters are represented as ordinal 10, or hexadecimal 0x0A.

Note: Do not use carriage returns (CR) as is the convention in Apple OS's (0x0D) or the carriage return - linefeed combination (CRLF) as is standard for the Windows OS (0x0D, 0x0A).

Naming Conventions

Classes (http://framework.zend.com/manual/en/coding-standard.naming-conventions.html)

The names of the classes directly map to the directories in which they are stored. All classes are stored hierarchically under the agr directory.

Class names may only contain alphanumeric characters. Numbers are permitted in class names but are discouraged in most cases. Underscores are only permitted in place of the path separator; the filename "*Zend/Db/Table.php*" must map to the class name "Zend_Db_Table".

If a class name is comprised of more than one word, the first letter of each new word must be capitalized. Successive capitalized letters are not allowed, e.g. a class "Zend_PDF" is not allowed while "Zend_Pdf" is acceptable.

Abstract Classes

In general, abstract classes follow the same conventions as classes, with one additional rule: abstract class names must end in the term, "Abstract", and that term must not be preceded by an underscore. As an example, Zend_Controller_Plugin_Abstract is considered an invalid name, but Zend_Controller_PluginAbstract or Zend_Controller_Plugin_PluginAbstract would be valid names.

Note: This naming convention is new with version 1.9.0 of Zend Framework. Classes that pre-date that version may not follow this rule, but will be renamed in the future in order to comply. The rationale for the change is due to namespace usage. As we look towards Zend Framework 2.0 and usage of PHP 5.3, we will be using namespaces. The easiest way to automate conversion to namespaces is to simply convert underscores to the namespace separator -- but under the old naming conventions, this leaves the classname as simply "Abstract" or "Interface" -- both of which are reserved keywords in PHP. If we prepend the (sub)component name to the classname, we can avoid these issues.
To illustrate the situation, consider converting the class Zend_Controller_Request_Abstract to use namespaces:

```
namespace Zend\Controller\Request;

abstract class Abstract
{
    // ...
}
```

Clearly, this will not work. Under the new naming conventions, however, this would become:

```
namespace Zend\Controller\Request;

abstract class RequestAbstract
{
    // ...
```

```
}
```

We still retain the semantics and namespace separation, while omitting the keyword issues; simultaneously, it better describes the abstract class.

Interfaces

In general, interfaces follow the same conventions as classes, with one additional rule: interface names may optionally end in the term, "Interface", but that term must not be preceded by an underscore. As an example, Zend_Controller_Plugin_Interface is considered an invalid name, but Zend_Controller_PluginInterface or Zend_Controller_Plugin_PluginInterface would be valid names.

While this rule is not required, it is strongly recommended, as it provides a good visual cue to developers as to which files contain interfaces rather than classes.

Note: This naming convention is new with version 1.9.0 of Zend Framework. Classes that pre-date that version may not follow this rule, but will be renamed in the future in order to comply. See the previous section for more information on the rationale for this change.

Filenames

For all other files, only alphanumeric characters, underscores, and the dash character ("-") are permitted. Spaces are strictly prohibited.

Any file that contains PHP code should end with the extension "*.php*", with the notable exception of view scripts.

The following examples show acceptable filenames:

```
views/header.php
Login/Login.php
```

File names must map to class names as described above.

Files which implement one of the standardized screens (see authorizedToBeHere.xlsx) must use the standardized name and be placed in a folder with the standardized name.

Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in function names but are discouraged in most cases.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called "camelCase" formatting.

Verbosity is generally encouraged. Function names should be as verbose as is practical to fully describe their purpose and behavior.

These are examples of acceptable names for functions:

```
filterInput()
getElementById()
widgetFactory()
```

For object-oriented programming, accessors for instance or static variables should always be prefixed with "get" or "set". In implementing design patterns, such as the singleton or factory patterns, the name of the method should contain the pattern name where practical to more thoroughly describe behavior.

For methods on objects that are declared with the "private" or "protected" modifier, the first character of the method name must be an underscore. This is the only acceptable application of an underscore in a method name. Methods declared "public" should never contain an underscore.

Functions in the global scope (a.k.a "floating functions") are permitted but discouraged in most cases. Consider wrapping these functions in a static class.

Variables

Variable names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in variable names but are discouraged in most cases.

For instance variables that are declared with the "private" or "protected" modifier, the first character of the variable name must be a single underscore. This is the only acceptable application of an underscore in a variable name. Member variables declared "public" should never start with an underscore.

As with function names (see section 3.3) variable names must always start with a lowercase letter and follow the "camelCaps" capitalization convention.

Verbosity is generally encouraged. Variables should always be as verbose as practical to describe the data that the developer intends to store in them. Terse variable names such as "*$i*" and "*$n*" are discouraged for all but the smallest loop contexts. If a loop contains more than 20 lines of code, the index variables should have more descriptive names.

Variables that hold an array should end with the word ‘Array’. Similarly for variables that hold other structures.

Constants

Constants may contain both alphanumeric characters and underscores. Numbers are permitted in constant names.

All letters used in a constant name must be capitalized, while all words in a constant name must be separated by underscore characters.

For example, **`EMBED_SUPPRESS_EMBED_EXCEPTION`** is permitted but **`EMBED_SUPPRESSEMBEDEXCEPTION`** is not.

Constants must be defined as class members with the "const" modifier. Defining constants in the global scope with the "define" function is permitted but strongly discouraged.

Coding Style (http://framework.zend.com/manual/en/coding-standard.coding-style.html)

PHP Code Demarcation

PHP code must always be delimited by the full-form, standard PHP tags:

```
<?php

?>
```

Short tags are never allowed. For files containing only PHP code, the closing tag must always be omitted (See General standards).

Strings

String Literals

When a string is literal (contains no variable substitutions), the apostrophe or "single quote" should always be used to demarcate the string:

```
$a = 'Example String';
```

String Literals Containing Apostrophes

When a literal string itself contains apostrophes, it is permitted to demarcate the string with quotation marks or "double quotes". This is especially useful for **`SQL`** statements:

```
$sql = "SELECT `id`, `name` from `people` "
     . "WHERE `name`='Fred' OR `name`='Susan'";
```

This syntax is preferred over escaping apostrophes as it is much easier to read.

Variable Substitution

Variable substitution is permitted using either of these forms:

```
$greeting = "Hello $name, welcome back!";
$greeting = "Hello {$name}, welcome back!";
```

For consistency, this form is not permitted:

```
$greeting = "Hello ${name}, welcome back!";
```

String Concatenation

Strings must be concatenated using the "." operator. A space must always be added before and after the "." operator to improve readability:

```
$company = 'Zend' . ' ' . 'Technologies';
```

When concatenating strings with the "." operator, it is encouraged to break the statement into multiple lines to improve readability. In these cases, each successive line should be padded with white space such that the "."; operator is aligned under the "=" operator:

```
$sql = "SELECT `id`, `name` FROM `people` "
     . "WHERE `name` = 'Susan' "
     . "ORDER BY `name` ASC ";
```

Arrays

Numerically Indexed Arrays

Negative numbers are not permitted as indices.

An indexed array may start with any non-negative number, however all base indices besides 0 are discouraged.

When declaring indexed arrays with the Array function, a trailing space must be added after each comma delimiter to improve readability:

```
$sampleArray = array(1, 2, 3, 'Zend', 'Studio');
```

It is permitted to declare multi-line indexed arrays using the "array" construct. In this case, each successive line must be padded with spaces such that beginning of each line is aligned:

```
$sampleArray = array(1, 2, 3, 'Zend', 'Studio',
```

```
                        $a, $b, $c,
                        56.44, $d, 500);
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration:

```
$sampleArray = array(
    1, 2, 3, 'Zend', 'Studio',
    $a, $b, $c,
    56.44, $d, 500,
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

Associative Arrays

When declaring associative arrays with the Array construct, breaking the statement into multiple lines is encouraged. In this case, each successive line must be padded with white space such that both the keys and the values are aligned:

```
$sampleArray = array('firstKey'  => 'firstValue',
                     'secondKey' => 'secondValue');
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration. For readability, the various "=>" assignment operators should be padded such that they align.

```
$sampleArray = array(
    'firstKey'  => 'firstValue',
    'secondKey' => 'secondValue',
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

Classes

Class Declaration

Classes must be named according to Zend Framework's naming conventions.

The brace should always be written on the line underneath the class name.

Every class must have a documentation block that conforms to the PHPDocumentor standard.

All code in a class must be indented with four spaces.

Only one class is permitted in each PHP file.

Placing additional code in class files is permitted but discouraged. In such files, two blank lines must separate the class from any additional PHP code in the class file.

The following is an example of an acceptable class declaration:

```
/**
* Documentation Block Here
*/
class SampleClass
{
    // all contents of class
    // must be indented four spaces
}
```

Classes that extend other classes or which implement interfaces should declare their dependencies on the same line when possible.

```
class SampleClass extends FooAbstract implements
BarInterface
{
}
```

If as a result of such declarations, the line length exceeds the maximum line length, break the line before the "extends" and/or "implements" keywords, and pad those lines by one indentation level.

```
class SampleClass
    extends FooAbstract
    implements BarInterface
{
```

```
}
```

If the class implements multiple interfaces and the declaration exceeds the maximum line length, break after each comma separating the interfaces, and indent the interface names such that they align.

```
class SampleClass
    implements BarInterface,
               BazInterface
{
}
```

Class Member Variables

Member variables must be named according to Zend Framework's variable naming conventions.

Any variables declared in a class must be listed at the top of the class, above the declaration of any methods.

The *var* construct is not permitted. Member variables always declare their visibility by using one of the private, protected, or public modifiers. Giving access to member variables directly by declaring them as public is permitted but discouraged in favor of accessor methods (set & get).

Functions and Methods

Function and Method Declaration

Functions must be named according to Zend Framework's function naming conventions.

Methods inside classes must always declare their visibility by using one of the private, protected, or public modifiers.

As with classes, the brace should always be written on the line underneath the function name. Space between the function name and the opening parenthesis for the arguments is not permitted.

Functions in the global scope are strongly discouraged.

The following is an example of an acceptable function declaration in a class:

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
```

```
     */
    public function bar()
    {
        // all contents of function
        // must be indented four spaces
    }
}
```

In cases where the argument list exceeds the maximum line length, you may introduce line breaks. Additional arguments to the function or method must be indented one additional level beyond the function or method declaration. A line break should then occur before the closing argument paren, which should then be placed on the same line as the opening brace of the function or method with one space separating the two, and at the same indentation level as the function or method declaration. The following is an example of one such situation:

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar($arg1, $arg2, $arg3,
        $arg4, $arg5, $arg6
    ) {
        // all contents of function
        // must be indented four spaces
    }
}
```

Note: *Note*: Pass-by-reference is the only parameter passing mechanism permitted in a method declaration.

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar(&$baz)
    {}
}
```

Call-time pass-by-reference is strictly prohibited.

The return value must not be enclosed in parentheses. This can hinder readability, in additional to breaking code if a method is later changed to return by reference.

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * WRONG
     */
    public function bar()
    {
        return($this->bar);
    }

    /**
     * RIGHT
     */
    public function bar()
    {
        return $this->bar;
    }
}
```

Function and Method Usage

Function arguments should be separated by a single trailing space after the comma delimiter. The following is an example of an acceptable invocation of a function that takes three arguments:

```
threeArguments(1, 2, 3);
```

Call-time pass-by-reference is strictly prohibited. See the function declarations section for the proper way to pass function arguments by-reference.

In passing arrays as arguments to a function, the function call may include the "array" hint and may be split into multiple lines to improve readability. In such cases, the normal guidelines for writing arrays still apply:

```
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'Zend', 'Studio',
                     $a, $b, $c,
                     56.44, $d, 500), 2, 3);

threeArguments(array(
    1, 2, 3, 'Zend', 'Studio',
```

```
    $a, $b, $c,
    56.44, $d, 500
), 2, 3);
```

Control Statements

If/Else/Elseif

Control statements based on the *if* and *elseif* constructs must have a single space before the opening parenthesis of the conditional and a single space after the closing parenthesis.

Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping for larger conditional expressions.

The opening brace is written on the same line as the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented using four spaces.

```
if ($a != 2) {
    $a = 2;
}
```

If the conditional statement causes the line length to exceed the maximum line length and has several clauses, you may break the conditional into multiple lines. In such a case, break the line prior to a logic operator, and pad the line such that it aligns under the first character of the conditional clause. The closing paren in the conditional will then be placed on a line with the opening brace, with one space separating the two, at an indentation level equivalent to the opening control statement.

```
if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d)
) {
    $a = $d;
}
```

The intention of this latter declaration format is to prevent issues when adding or removing clauses from the conditional during later revisions.

For "if" statements that include "elseif" or "else", the formatting conventions are similar to the "if" construct. The following examples demonstrate proper formatting for "if" statements with "else" and/or "elseif" constructs:

```
if ($a != 2) {
    $a = 2;
} else {
```

```
    $a = 7;
}

if ($a != 2) {
    $a = 2;
} elseif ($a == 3) {
    $a = 4;
} else {
    $a = 7;
}

if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d)
) {
    $a = $d;
} elseif (($a != $b)
          || ($b != $c)
) {
    $a = $c;
} else {
    $a = $b;
}
```

PHP allows statements to be written without braces in some circumstances. This coding standard makes no differentiation- all "if", "elseif" or "else" statements must use braces.

Switch

Control statements written with the "switch" statement must have a single space before the opening parenthesis of the conditional statement and after the closing parenthesis.

All content within the "switch" statement must be indented using four spaces. Content under each "case" statement must be indented using an additional four spaces.

```
switch ($numPeople) {
    case 1:
        break;

    case 2:
        break;

    default:
        break;
}
```

The construct default should never be omitted from a switch statement.

Note: *Note*: It is sometimes useful to write a case statement which falls through to the next case by not including a break or return within that case. To distinguish these cases from bugs, any case statement where break or return are omitted should contain a comment indicating that the break was intentionally omitted.

Other Control Structures

```
Format loops similarly to how if statements were formatted.

for($i = 0; $i < $count; $i++) {
    // your code here
}

// same format as the if statement
while($i < $n) {
    // your code here
}

foreach ($value as $k => $v) {
    // your code here
}
```

Inline Documentation

Documentation Format

All documentation blocks ("docblocks") must be compatible with the phpDocumentor format. Describing the phpDocumentor format is beyond the scope of this document. For more information, visit: » http://phpdoc.org/

All class files must contain a "file-level" docblock at the top of each file and a "class-level" docblock immediately above each class. Examples of such docblocks can be found below.

Files

Every file that contains PHP code must have a docblock at the top of the file that contains these phpDocumentor tags at a minimum:

```
/**
* Short description for file
* Long description for file (if any)...
*
*/
```

Classes

Every class must have a docblock that contains these phpDocumentor tags at a minimum:

```
/**
* Short description for class
*
* Long description for class (if any)...
*
*/
```

Functions

Every function, including object methods, must have a docblock that contains at a minimum:

- A description of the function
- All of the arguments
- All of the possible return values

It is not necessary to use the "@access" tag because the access level is already known from the "public", "private", or "protected" modifier used to declare the function.

If a function or method may throw an exception, use @throws for all known exception classes:

```
@throws exceptionclass [description]
```

JavaScript coding standards (http://drupal.org/node/172169)

Indenting

Use an indent of 2 spaces, with no tabs. No trailing whitespace.

String Concatenation

Always use a space between the + and the concatenated parts to improve readability.

```
var string = 'Foo' + bar;
string = bar + 'foo';
string = bar() + 'foo';
string = 'foo' + 'bar';
```

When using the concatenating assignment operator ('+='), use a space on each side as with the assignment operator:

```
var string += 'Foo';
string += bar;
string += baz();
```

CamelCasing

Unlike the variables and functions defined in Drupal's PHP, multi-word variables and functions in JavaScript should be lowerCamelCased. The first letter of each variable or function should be lowercase, while the first letter of subsequent words should be capitalized. There should be no underscores between the words.

Semi-colons

JavaScript allows any expression to be used as a statement and uses semi-colons to mark the end of a statement. However, it attempts to make this optional with "semi-colon insertion", which can mask some errors and will also cause JS aggregation to fail. All statements should be followed by ; except for the following:

```
for, function, if, switch, try, while
```

The exceptions to this are functions declared like

```
  Drupal.behaviors.tableSelect = function (context) {
    // Statements...
  };
```

and

```
  do {
    // Statements...
  } while (condition);
```

These should all be followed by a semi-colon.

In addition the `return` value expression must start on the same line as the return keyword in order to avoid semi-colon insertion.

If the "Optimize JavaScript files" performance option in Drupal 6 is enabled, and there are missing semi-colons, then the JS aggregation will fail. It is therefore very important that semi-colons are used.

Control Structures

These include `if, for, while, switch`, etc. Here is an example `if` statement, since it is the most complicated of them:

```
if (condition1 || condition2) {
  action1();
}
else if (condition3 && condition4) {
  action2();
}
else {
  defaultAction();
}
```

Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.

You are strongly encouraged to always use curly braces even in situations where they are technically optional. Having them increases readability and decreases the likelihood of logic errors being introduced when new lines are added.

switch

For `switch` statements:

```
switch (condition) {
  case 1:
    action1();
    break;

  case 2:
    action2();
    break;

  default:
```

```
    defaultAction();
}
```

try

The `try` class of statements should have the following form:

```
  try {
    // Statements...
  }
  catch (variable) {
    // Error handling...
  }
  finally {
    // Statements...
  }
```

for in

The `for in` statement allows for looping through the names of all of the properties of an object. Unfortunately, all of the members which were inherited through the prototype chain will also be included in the loop. This has the disadvantage of serving up method functions when the interest is in data members. To prevent this, the body of every `for in` statement should be wrapped in an `if` statement that does filtering. It can select for a particular type or range of values, or it can exclude functions, or it can exclude properties from the prototype. For example:

```
for (var variable in object) if (filter) {
  // Statements...
}
```

You can use the `hasOwnProperty` method to distinguish the true members of the object:

```
for (var variable in object) if (object.hasOwnProperty(variable)) {
  // Statements...
}
```

Functions

Functions and Methods

Functions and methods should be named using lowerCamelCase.

```
Drupal.behaviors.tableDrag = function (context) {
  for (var base in Drupal.settings.tableDrag) {
    if (!$('#' + base + '.tabledrag-processed', context).size()) {
      $('#' + base).filter(':not(.tabledrag-processed)').each(addBehavior);
      $('#' + base).addClass('tabledrag-processed');
    }
  }
};
```

Function Calls

Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon. Here's an example:

```
foobar = foo(bar, baz, quux);
```

As displayed above, there should be one space on either side of an equals sign used to assign the return value of a function to a variable. In the case of a block of related assignments, more space may be inserted to promote readability:

```
short        = foo(bar);
longVariable = foo(baz);
```

If a function literal is anonymous, there should be one space between the word function and the left parenthesis. If the space is omitted, then it can appear that the function's name is actually "function".

```
div.onclick = function (e) {
  return false;
};
```

Function Declarations

```
function funStuff(field) {
  alert("This JS file does fun message popups.");
  return field;
}
```

Arguments with default values go at the end of the argument list. Always attempt to return a meaningful value from a function if one is appropriate.

Please note the special notion of anonymous functions explained above.

Variables and Arrays

All variables should be declared with `var` before they are used and should only be declared once. Doing this makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals.

Variables should not be defined in the global scope; try to define them in a local function scope at all costs. All variables should be declared at the beginning of a function.

Constants and Global Variables

lowerCamelCasing should be used for pre-defined constants. Unlike the PHP standards, you should use lowercase `true`, `false` and `null` as the uppercase versions are not valid in JS.

Variables added through `drupal_add_js()` should also be lowerCamelCased, so that they can be consistent with other variables once they are used in JavaScript.

```
<?php
drupal_add_js(array('myModule' => array('basePath' => base_path())),
'setting');
?>
```

This variable would then be referenced:

```
Drupal.settings.myModule.basePath;
```

Arrays

Arrays should be formatted with a space separating each element and assignment operator, if applicable:

```
someArray = ['hello', 'world'];
```

Note that if the line spans longer than 80 characters (often the case with form and menu declarations), each element should be broken into its own line, and indented one level:

Note there is no comma at the end of the last array element. This is different from the PHP coding standards.. Having a comma on the last array element in JS will cause an exception to occur.

Comments

Inline documentation for source files should follow the Doxygen formatting conventions.

Non-documentation comments are strongly encouraged. A general rule of thumb is that if you look at a section of code and think "Wow, I don't want to try and describe that", you need to comment it before you forget how it works. Comments can be removed by JS compression utilities later, so they don't negatively impact on the file download size.

Non-documentation comments should use capitalized sentences with punctuation. All caps are used in comments only when referencing constants, e.g., TRUE. Comments should be on a separate line immediately before the code line or block they reference. For example:

```
// Unselect all other checkboxes.
```

If each line of a list needs a separate comment, the comments may be given on the same line and may be formatted to a uniform indent for readability.

C style comments (`/* */`) and standard C++ comments (`//`) are both fine.

Header Comment Blocks

All source code files in the core Drupal distribution should contain the following comment block as the header:

```
// $Id$
```

This tag will be expanded by the CVS to contain useful information

```
// $Id: CODING_STANDARDS.html,v 1.14 2008/02/19 03:36:41 ax Exp $
```

JS code placement

JavaScript code should not be embedded in the HTML where possible, as it adds significantly to page weight with no opportunity for mitigation by caching and compression.

"with" statement

The `with` statement was intended to provide a shorthand for accessing members in deeply nested objects. For example, it is possible to use the following shorthand (but not recommended) to access `foo.bar.foobar.abc`, etc:

```
with (foo.bar.foobar) {
  var abc = true;
  var xyz = true;
}
```

However it's impossible to know by looking at the above code which `abc` and `xyz` will get modified. Does `foo.bar.foobar` get modified? Or is it the global variables `abc` and `xyz`?

Instead you should use the explicit longer version:

```
foo.bar.foobar.abc = true;
foo.bar.foobar.xyz = true;
```

or if you really want to use a shorthand, use the following alternative method:

```
  var o = foo.bar.foobar;
  o.abc = true;
  o.xyz = true;
```

Operators

True or false comparisons

The `==` and `!=` operators do type coercion before comparing. This is bad because it causes:
`' \t\r\n' == 0`
to be true. This can mask type errors. When comparing to any of the following values, use the `===` or `!==` operators, which do not do type coercion:
`0 '' undefined null false true`

Comma Operator

The comma operator causes the expressions on either side of it to be executed in left-to-right order, and returns the value of the expression on the right, and should be avoided. Example usage is:

```
var x = (y = 3, z = 9);
```

This sets x to 9. This can be confusing for users not familiar with the syntax and makes the code more difficult to read and understand. So avoid the use of the comma operator except for in the control part of `for` statements. This does not apply to the comma separator (used in object literals, array literals, etc.)

Avoiding unreachable code

To prevent unreachable code, a `return`, `break`, `continue`, or `throw` statement should be followed by a `}` or `case` or `default`.

Constructors

Constructors are functions that are designed to be used with the `new` prefix. The `new` prefix creates a new object based on the function's prototype, and binds that object to the function's implied this parameter. JavaScript doesn't issue compile-time warning or run-time warnings if a required `new` is omitted. If you neglect to use the `new` prefix, no new object will be made and this will be bound to the global object (bad). Constructor functions should be given names with an initial uppercase and a function with an initial uppercase name should not be called unless it has the `new` prefix.

Use literal expressions

Use literal expressions instead of the `new` operator:

- Instead of `new Array()` use `[]`
- Instead of `new Object()` use `{}`

- Don't use the wrapper forms `new Number`, `new String`, `new Boolean`.

In most cases, the wrapper forms should be the same as the literal expressions. However, this isn't always the case, take the following as an example:

```
var literalNum = 0;
var objectNum = new Number(0);
if (literalNum) { } // false because 0 is a false value, will not be
executed.
if (objectNum) { }  // true because objectNum exists as an object, will be
executed.
if (objectNum.valueOf()) { } // false because the value of objectNum is 0.
```

eval is evil

`eval()` is evil. It effectively requires the browser to create an entirely new scripting environment (just like creating a new web page), import all variables from the current scope, execute the script, collect the garbage, and export the variables back into the original environment. Additionally, the code cannot be cached for optimization purposes. It is probably the most powerful and most misused method in JavaScript. It also has aliases. So do not use the `Function` constructor and do not pass strings to `setTimeout()` or `setInterval()` .

Preventing XSS

All output to the browser that has been provided by a user should be run through the `Drupal.checkPlain()` function first. This is similar to Drupal's PHP `check_plain()` and encodes special characters in a plain-text string for display as HTML.

Typeof

When using a `typeof` check, don't use the parenthesis for the typeof. The following is the correct coding standard:

```
if (typeof myVariable == 'string') {
  // ...
}
```

Modifying the DOM

When adding new HTML elements to the DOM, don't use `document.createElement()`. For cross-browser compatibility reasons and also in an effort to reduce file size, you should use the jQuery equivalent.

Don't:

```
this.popup = document.createElement('div');
this.popup.id = 'autocomplete';
```

Do:

```
this.popup = $('<div id="autocomplete"></div>')[0];
```

SQL coding conventions (http://drupal.org/node/2497)

Don't use Reserved Words

Don't use (ANSI) SQL / MySQL / PostgreSQL / MS SQL Server / ... Reserved Words for column and/or table names. Even if this may work with your (MySQL) installation, it may not with others or with other databases. Some references:

- (ANSI) SQL Reserved Words
- MySQL Reserved Words: 5.1, 5.0, 3.23.x, 4.0, 4.1
- PostgreSQL Reserved Words
- Oracle Reserved Words (in particular UID is a problem in our context)
- MS SQL Server Reserved Words
- DB2 Reserved Words

Some commonly misused keywords: `TIMESTAMP, TYPE, TYPES, MODULE, DATA, DATE, TIME, ...`
See also [bug] SQL Reserved Words.

Use PHP Data Objects

Capitalization and user-supplied data

- Make SQL reserved words UPPERCASE.
- Make column and constraint names lowercase.

Naming

- Use singular nouns for table names since they describe the entity the table represents.
- Name every constraint (primary, foreign, unique keys) yourself.
- Index names should begin with the name of the table they depend on, eg. `INDEX users_sid_idx`.

Indentation

Possible ways to indent or format longer SQL queries on multiple lines:

```
<?php
if (!(db_query(
  "
    INSERT INTO {mlsp_media_file_type}
    SET extension   = '%s',
    attributes      = '%s'
  ",
```

```
  $file_type_entry['extension'],
  $selected_attributes
))) {
  $errors = TRUE;
}
?>
```

or

```
<?php
$sql = "SELECT t.*, j1.col1, j2.col2"
. " FROM {table} AS t"
. " LEFT JOIN {join1} AS j1 ON j1.id = t.jid"
. " LEFT JOIN {join2} AS j2 ON j2.id = t.jjid"
. " WHERE t.col LIKE '%s'"
. " ORDER BY %s"
;
$result = db_query($sql, 'findme', 't.weight');
?>
```

- Avoid "SELECT * FROM ..."

```
<?php
$sql = "
  SELECT    t.*
  ,         j1.col1
  ,         j2.col2
  FROM      {table} AS t
  LEFT JOIN {join1} AS j1 ON j1.id = t.jid
  LEFT JOIN {join2} AS j2 ON j2.id = t.jjid
  WHERE     t.col LIKE '%s'
  AND       t.status = 1
  AND       t.type = 'foobar'
  ORDER BY  t.bogus ASC
  ,         t.bar DESC
  LIMIT     10, 30
";

$result = db_query($sql, 'findme');
?>
```

Another Example:

```
<?php
$sql = "
  SELECT    t.*
  ,         j1.col1
  ,         j2.col2
  -- ,         j2.col3
  FROM      {table} AS t
  LEFT JOIN {join1} AS j1 ON j1.id = t.jid
  LEFT JOIN {join2} AS j2 ON j2.id = t.jjid
  WHERE     t.col LIKE '%s'
  AND       t.status = 1
  AND       t.type = 'foobar'
  ORDER BY  t.bogus ASC
```

```
  -- ,          t.bar DESC
  LIMIT      10, 30
";

$result = db_query($sql, 'findme');
?>
```

Another Example:

```
<?php
$sql = "
  SELECT     t.*
  ,          j1.col1
  ,          j2.col2
  ,          j2.col3
  ,          (
    SELECT     b.dateme
    FROM       {date_persons} AS dp
    WHERE      dp.person_id = t.person_id
    LIMIT      1
  ) AS dateme
  FROM       {table} AS t
  LEFT JOIN {join1} AS j1 ON j1.id = t.jid
  LEFT JOIN {join2} AS j2 ON j2.id = t.jjid
  WHERE      t.col LIKE '%s'
  AND        t.status = 1
  AND        t.type = 'foobar'
  AND        t.person_id IN(
    SELECT     t.id
    FROM       {persons}
    WHERE      t.name LIKE '%s%%'
  )
  ORDER BY   t.bogus ASC
  ,          t.bar DESC
  LIMIT      10, 30
";

$result = db_query($sql, 'findme', 'A');
?>
```

HTML Coding Style

(http://www.orangehrm.com/wiki/index.php/HTML_Coding_Conventions)

All HTML tags should be in lower case

Eg:

```
<table></table>
<form></form>
```

Use meaningful names for ID's and Names of HTML elements

Use meaningful ID's and names for HTML elements Eg:

```
txtName, txtAge
```

instead of

```
text1, text2
```

Indent HTML code consistently

Indent code consistently. eg:

```
<body>
    <form name="frmActivity" method="post" action="<?php echo
$formAction;?>">
        <input type="hidden" name="sqlState" value="">
        <input type="hidden" name="delState" value="">
        <input type="hidden" name="activityId" value="">
        <label for="cmbProjectId"></label>
        <select name="cmbProjectId">
            <option value="value1"></option>
        </select>
    </form>
</body>
```

Maximum line length

Limit maximum line length to around 120 characters. Wrap code if longer than that. This will improve readability of code.

Eg: Wrapping code to limit line length

```
<div id ="addActivityLayer">
    <img onClick="<?php echo $saveBtnAction; ?>;"
        onMouseOut="this.src='../../themes/beyondT/pictures/btn_save.gif';"
```

```
onMouseOver="this.src='../../themes/beyondT/pictures/btn_save_02.gif';"
        src="../../themes/beyondT/pictures/btn_save.gif"/>
</div>
```

Only use tables for tabular data, use CSS for styling

Do not use tables for layout, use CSS instead.

Do not heavily comment HTML code

Adding too much HTML comments adds to the overhead in bandwidth. Therefore comment sparingly in HTML and use descriptive element names to improve readability.

Do not use inline style attributes

As much as possible, use css classes instead of using inline style attributes. This has the advantages:

- Makes it easier to change the look of the UI by changing the stylesheet
- Makes it possible to reuse the css class in another place, giving a more consistent UI and making it possible to change the look by changing only one place.

```
<!-- INSTEAD OF -->
<div style="width:100px;align:center;">

<!-- USE -->
<div class="message">
```

Use external CSS files instead of style tags in the HTML page

Use external css files, instead of using inline css in the HTML page by using <style> tags. This gives the same advantages as in #Do not Use inline style attributes above.

Refer to images from CSS files when possible

Referring to images from CSS files instead of directly using from the HTML code makes it easier to change the look of the application by changing stylesheets.

All HTML code should validate as XHTML 1.0

Document type

Use the following document type

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

How to validate

You can easily validate html in OrangeHRM using Firefox's "Web developer toolbar". First open just the main frame (without the top level menus etc. - "right click - this frame -> show only this frame" in firefox) and use the 'Tools->Validate Local HTML" option in the web developer toolbar.

Some common issues to look out for

1. Tables, cells not closed properly
2. Elements not closed eg:

   ```
   <input type="text >
   ```

   Fixed by changing to:

   ```
   <input type="text" />
   ```
3. Attributes not quoted: eg:

   <option value=0>Select</option>

   Fixed by changing to:

   <option value="0">Select</option>
4. height attribute in tables:

   ```
   <table height="100">
   ```

   If height is needed, can be converted to a style: eg:

   ```
   <table style="height:100px">
   ```

   Note that external css is preferred to inline style attribute. So the preferred solution would be:

   ```
   <table class="abc">
   ```

   and a css rule

   ```
   table.abc {height: 100px;}
   ```
5. Tables with empty bodies also do not validate. eg:

6. <table></table> or

<table><tbody></tbody></table>

Therefore, make sure that the table has some rows or is completely omitted, in the case where there is no data

7. <script> tag for inline javascript should be as follows:

   <script type="text/javascript">
   //<![CDATA[
   function test() {
   }
   //]]>
   </script>

   Note that
   `<script language="Javascript">`
   which we have in some places is not valid.

8. All url's in HTML (not in javscript code) should use & instead of &

   eg:
   `./lib/controllers/CentralController.php?uniqcode=EMP&amp;VIEW=MAIN`

9. All <img tags should have the alt attribute. You can use alt="" for images used just for style.
10. <textarea> should have cols and rows attributes. eg:

    `<textarea cols="25" rows="3" name="xx"></textarea>`

CSS coding standards (http://drupal.org/node/302199)

Write valid CSS

All CSS code should be valid CSS, preferably to CSS 2.1. CSS 3.0 is acceptable too, provided the usage can be justified and the principles of graceful degradation / progressive enhancement are followed.

Concise terminology used in these standards:

```
selector {
  property: value;
}
```

Selectors

Selectors should

- be on a single line
- have a space after the previous selector
- end in an opening brace
- be closed with a closing brace on a separate line without indentation

```
.book-navigation .page-next {
}
.book-navigation .page-previous {
}

.book-admin-form {
}
```

A blank line should be placed between each group, section, or block of multiple selectors of logically related styles.

Multiple selectors

Multiple selectors should each be on a single line, with no space after each comma:

```
#forum td.posts,
#forum td.topics,
#forum td.replies,
#forum td.pager {
```

Properties

All properties should be on the following line after the opening brace. Each property should:

- be on its own line
- be indented one tab space
- have no space between the property name and the colon and a have one space after the colon before the property value
- end in a semi-colon

```
#forum .description {
  color: #EFEFEF;
  font-size: 0.9em;
  margin: 0.5em;
}
```

Alphabetizing properties

Multiple properties should be listed in alphabetical order.

Note that colors in RGB notation (e.g., #FFF) are preferred in uppercase, whereas non-color values should be in lowercase.

Properties with multiple values

Where properties can have multiple values, each value should be separated with a space.

```
  font-family: helvetica, sans-serif;
```

Comments

In line with PHP coding standards, block level documentation should be used as follows, to describe a section of code below the comment.

```
/**
 * Documentation here.
 */
```

Every CSS file should have a CVS ID tag and file header at the top:

```
/* $Id$ */

/**
 * @file
 * Styles for [purpose].
 *
 * [If required, continue short summary above with longer explanation as
 * description, wrapping at 80 chars.]
```

```
 */
```

Shorter inline comments may be added after a property, preceded with a space:

```
  background-position: 0.2em 0.2em; /* LTR */
  padding-left: 2em; /* LTR */
```

Right-To-Left

Conditional loading of CSS files with specific override rules for right-to-left languages is allowed. For a module, the override rule should be defined in a file named <module>-rtl.css (e.g., node-rtl.css). For a theme, the override rule should be defined in a file named style-rtl.css. The rule that is overridden should be commented in the default CSS rule.

From node-rtl.css:

```
#node-admin-buttons {
  clear: left;
  float: right;
  margin-left: 0;
  margin-right: 0.5em;
}
```

Rules in node.css which will be overridden if the rtl.css file is loaded:

```
#node-admin-buttons {
  clear: right; /* LTR */
  float: left; /* LTR */
  margin-left: 0.5em; /* LTR */
}
```

See also: http://drupal.org/node/132442#language-rtl

As a rule of thumb, add a `/* LTR */` comment in your style:

- when you use the keywords, 'left' or 'right' in a property, e.g. `float: left;`
- where you use unequal margin, padding or borders on the sides of a box, e.g.,
- `margin-left: 1em;`
- `margin-right: 2em;`
- where you specify the direction of the language, e.g. `direction: ltr;`