Best Practices: PHP Coding Style

By Tim Perdue - PHPBuilder.com

One of PHP's greatest strengths can also be a great weakness in the wrong hands. I'm talking about its forgiving nature. One of the reasons why PHP has become so wildly popular is because it lets very inexperienced web developers build powerful applications without much planning, consistency, or documentation.

Unfortunately, that means an awful lot of PHP code out there is sloppy, hard to read and impossible to maintain. I know because I've written a lot of it ;-).

To address this and a lot of other issues, a number of the core PHP developers and community members got together and started the PEAR Project, which stands for PHP Extension and Add-on Repository. To date, the docs and other fruit from the PEAR project have been pretty sparse and difficult to follow, so this article is an attempt to shed some light on what they're doing.

A pretty huge part of maintainability of code is how it is formatted and commented. All code throughout a given project should be formatted the same way. I'm a pretty big stickler about this in the SourceForge codebase and you should be too.

Indenting

All your code should be properly indented. This is the most fundamental thing you can do to improve readability. Even if you don't comment your code, indenting will be a big help to anyone who has to read your code after you.

```
while ($x < $z) {
        if ($a == 1) {
                echo 'A was equal to 1';
        } else {
                if ($b == 2) {
                        //do something
                } else {
                        //do something else
                }
        }
}
```

The PEAR RFC standard calls for 4 spaces, not tabs of any size, in your code. I disagree with this personally and will continue to tab my code. My view is that tabs rather than spaces will create smaller files and smaller files are faster to parse, upload, download, etc etc. The other advantage to using tabs is that you can set your tab size to your personal preference when viewing someone else's code. I used to use 8-space tabs, but recently switched to 4-space tabs and all my code "reformatted" automatically by just setting a preference in vim.

Control Structures

This is pretty much common sense, but I still see an awful lot of code that is not braced for readability. If you use conditional expressions (IF statements) without braces, not only is it less readable, but bugs can also be introduced when someone modifies your code.

Bad Example:

```
if ($a == 1) echo 'A was equal to 1';
```

That's pretty much illegible. It may work for you, but the person following after you won't appreciate it at all.

Less Bad Example:

```
if ($a == 1)
        echo 'A was equal to 1';
```

Now that's at least readable, but it's still not maintainable. What if I want an additional action to occur when $a==1? I need to add braces, and if I forget to, I'll have a bug in my code.

Correct:

```
if (($a == 1) && ($b==2)) {
        echo 'A was equal to 1';
        //easily add more code
} elseif (($a == 1) && ($b==3)) {
        //do something
}
```

Notice the space after the **if** and **elseif** - this helps distinguish conditionals from function calls.

Function Calls

Functions should be called with no space between between the function name and the parentheses. Spaces between params and operators are encouraged.

```
$var = myFunction($x, $y);
```

Functions

Function calls are braced differently than conditional expressions. This is a case where I'll have to change my personal coding style in order to be kosher, but I guess it needs to be done.

```
function myFunction($var1, $var2 = '')
{
        //indent all code inside here
        return $result;
}
```

Notice again there is no space between the function name and the parens, and that the params are nicely spaced. All your code inside the function will be at least 4 spaces indented.

Another important principle when coding functions is that they should always **return instead of print directly**. Remember, if you print directly inside your function call, whomever calls your function cannot capture its output using a variable.

Comments

Borrowing - almost in its entirety - from the JavaDoc spec, PEAR strongly encourages the use of PHPDoc (www.phpdoc.de) comment style. JavaDoc is rather clever because you format your code comments in such a manner that they can be parsed by a doc-generating tool, essentially creating "self commenting" code. You can then view the resulting docs using a web browser.

As a reformed java programmer myself, I really like this standard and am trying to go back through my code and add PHPDoc comments.

```
/**
 *      short description of function
 *
 *      Optional more detailed description.
 *
 *      @param $paramName - type - brief purpose
 *      @param ...
 *      ...
 *      @return type and description
 */
```

Including Code

PHP4 introduced a pretty useful new feature: include_once(). I've seen a lot of questions on the mailing lists and discussion boards caused by people including the same files in multiple places. This can cause conflicts when the included files include functions, which can be defined only once per script.

The simple solution is to replace all your include() and require() calls with corresponding include_once () and require_once(). Both use the same file list, so a file included with require_once() will not be later included with an include_once() call. (technically, require_once and include_once are "operators", not "functions" so parens are not necessary).

I never use include, or any of these *_once functions. IMHO, well-designed applications include files in one place that is easy to maintain and keep track of. Others will disagree and say that each included file should require_once() every file that it depends on. This seems like extra overhead and maintenance headaches to me personally. Either way, the days of conflicting includes are probably over.

PHP Tags

You should always use the full <?php ?> tags to enclose your code, not the abbreviated <? ?> tags, which could conflict with XML or other languages. ASP-style tags, while supported, are messy and discouraged.

Strings

The proper use of strings is not really discussed in the PEAR RFC, but I'm going to cover it anyway. In PHP of course, double quotes (") are parsed, but single quotes (') are not. That means PHP does extra work (magic really) on double-quoted strings that it doesn't do on single-quoted strings. So there is a (subtle) performance difference between the two, especially in code that is iterated or called a large number of times.

Best:

```
$associative_array['name'];
$var='My String';
$var2='Very... long... string... ' . $var . ' ...more string... ';
$sql="INSERT INTO mytable (field) VALUES ('$var')";
```

Acceptable:

```
$associative_array["name_$x"];
$var="My String $a";
```

The first example does not include any vars that need to be appended or parsed into the strings, so single quotes (') are definitely the best way to go. The second example includes very short strings and has variables that need to be parsed into the strings, so it is acceptable to use double quotes. If the

strings were long, it would be best to use the append (.) operator.

For consistency's sake, I always use single quotes around all my strings, except SQL statements, which almost always contain apostrophes and variables that must be parsed into the strings.

Well, now you've got the fundamentals of good coding style to work with. If you have comments, please post them below. Not everyone will agree with the standards laid out here, nor should they. I may respectfully disagree with certain aspects of this RFC, but I strongly agree with the core value of it and its goals (to improve PHP coding for everyone).

--Tim