



CODE > HTML & CSS

# Top 15+ Best Practices for Writing Super Readable Code

by [Burak Guzel](#) 30 Mar 2011

Difficulty: Intermediate Languages: English ▾

HTML & CSS

Web Development

CSS

HTML



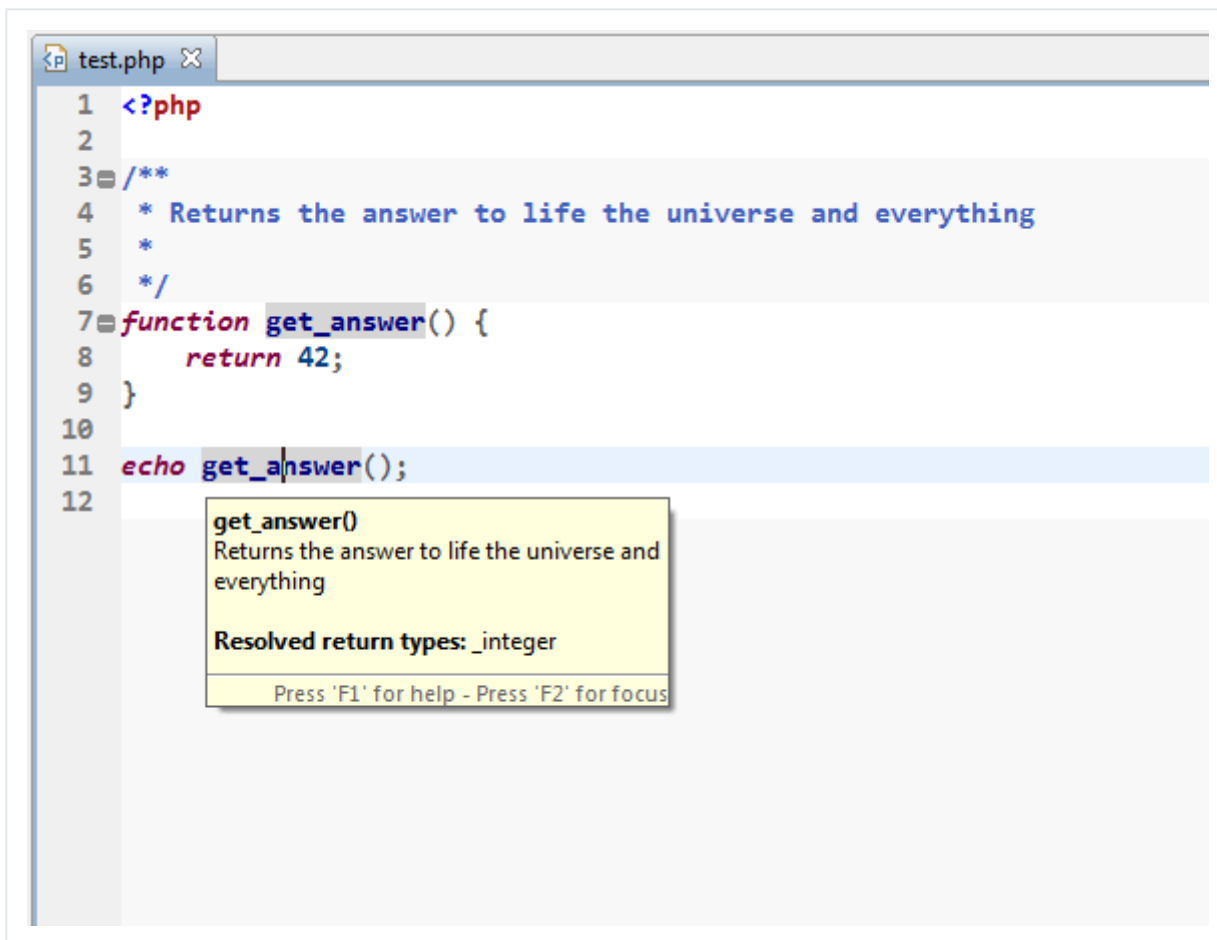
Twice a month, we revisit some of our readers' favorite posts from throughout the history of Nettuts+.

Code readability is a universal subject in the world of computer programming. It's one of the first things we learn as developers. This article will detail the fifteen most important best practices when writing readable code.

## 1 - Commenting & Documentation

IDE's (Integrated Development Environment) have come a long way in the past few years. This made commenting your code more useful than ever. Following certain standards in your comments allows IDE's and other tools to utilize them in different ways.

Consider this example:

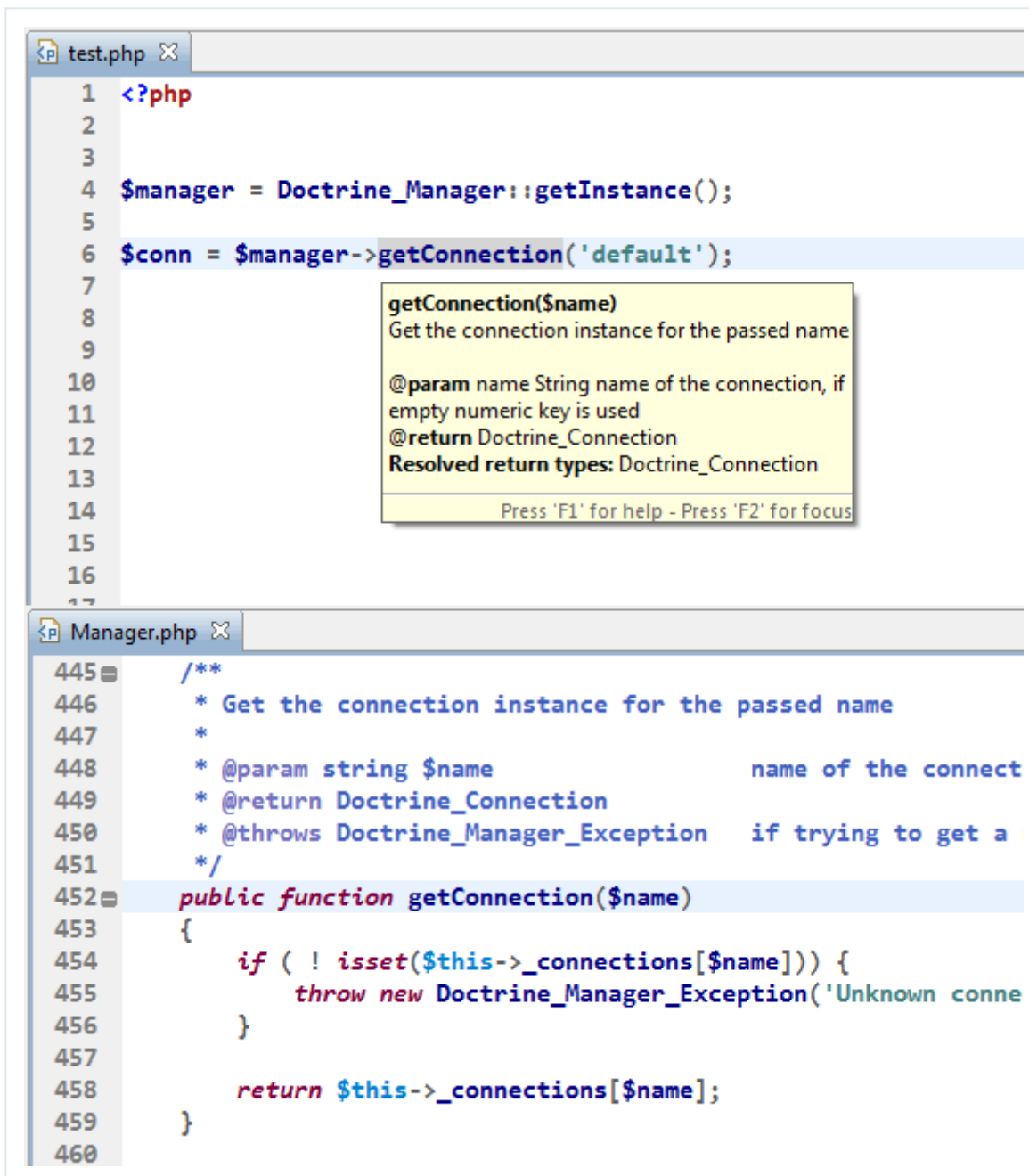


```
1 <?php
2
3 /**
4  * Returns the answer to life the universe and everything
5  *
6  */
7 function get_answer() {
8     return 42;
9 }
10
11 echo get_answer();
12
```

**get\_answer()**  
Returns the answer to life the universe and everything  
**Resolved return types:** \_integer  
Press 'F1' for help - Press 'F2' for focus

The comments I added at the function definition can be previewed whenever I use that function, even from other files.

Here is another example where I call a function from a third party library:



In these particular examples, the type of commenting (or documentation) used is based on [PHPDoc](#), and the IDE is [Aptana](#).

## 2 - Consistent Indentation

I assume you already know that you should indent your code. However, it's also worth noting that it is a good idea to keep your indentation style consistent.

There are more than one way of indenting code.

### Style 1:

```
1  function foo() {
2      if ($maybe) {
3          do_it_now();
4          again();
5      } else {
6          abort_mission();
7      }
8      finalize();
9  }
```

### Style 2:

```
01  function foo()
02  {
03      if ($maybe)
04      {
05          do_it_now();
06          again();
07      }
08      else
09      {
10          abort_mission();
11      }
12      finalize();
13  }
```

### Style 3:

```
01  function foo()
02  {  if ($maybe)
03      {  do_it_now();
04          again();
05      }
06      else
07      {  abort_mission();
08      }
09      finalize();
10  }
```

I used to code in style #2 but recently switched to #1. But that is only a matter of preference. There is no "best" style that everyone should be following. Actually, the best style, is a consistent style. If you are part of a team or if you are contributing code to a project, you should follow the existing style that is being used in that project.

The indentation styles are not always completely distinct from one another. Sometimes, they mix different rules. For example, in [PEAR Coding Standards](#), the opening bracket `"{"`

goes on the same line as [control structures](#), but they go to the next line after [function definitions](#).

PEAR Style:

```
01 function foo()  
02 {                               // placed on the next line  
03     if ($maybe) {              // placed on the same line  
04         do_it_now();  
05         again();  
06     } else {  
07         abort_mission();  
08     }  
09     finalize();  
10 }
```

Also note that they are using four spaces instead of tabs for indentations.

[Here](#) is a Wikipedia article with samples of different indent styles.

## 3 - Avoid Obvious Comments

Commenting your code is fantastic; however, it can be overdone or just be plain redundant. Take this example:

```
1 // get the country code  
2 $country_code = get_country_code($_SERVER['REMOTE_ADDR']);  
3  
4 // if country code is US  
5 if ($country_code == 'US') {  
6  
7     // display the form input for state  
8     echo form_input_state();  
9 }
```

When the text is that obvious, it's really not productive to repeat it within comments.

If you must comment on that code, you can simply combine it to a single line instead:

```
1 // display state selection for US users  
2 $country_code = get_country_code($_SERVER['REMOTE_ADDR']);
```

```
3 | if ($country_code == 'US') {  
4 |     echo form_input_state();  
5 | }
```

## 4 - Code Grouping

More often than not, certain tasks require a few lines of code. It is a good idea to keep these tasks within separate blocks of code, with some spaces between them.

Here is a simplified example:

```
01 | // get list of forums  
02 | $forums = array();  
03 | $r = mysql_query("SELECT id, name, description FROM forums");  
04 | while ($d = mysql_fetch_assoc($r)) {  
05 |     $forums []= $d;  
06 | }  
07 |  
08 | // load the templates  
09 | load_template('header');  
10 | load_template('forum_list',$forums);  
11 | load_template('footer');
```

Adding a comment at the beginning of each block of code also emphasizes the visual separation.

## 5 - Consistent Naming Scheme

PHP itself is sometimes guilty of not following consistent naming schemes:

- `strpos()` vs. `str_split()`
- `imagetypes()` vs. `image_type_to_extension()`

First of all, the names should have word boundaries. There are two popular options:

- **camelCase:** First letter of each word is capitalized, except the first word.
- **underscores:** Underscores between words, like: `mysql_real_escape_string()`.

Having different options creates a situation similar to the indent styles, as I mentioned earlier. If an existing project follows a certain convention, you should go with that. Also, some language platforms tend to use a certain naming scheme. For instance, in Java, most code uses camelCase names, while in PHP, the majority of uses underscores.

These can also be mixed. Some developers prefer to use underscores for procedural functions, and class names, but use camelCase for class method names:

```
01 | class Foo_Bar {  
02 |  
03 |     public function someDummyMethod() {  
04 |  
05 |     }  
06 |  
07 | }  
08 |  
09 | function procedural_function_name() {  
10 |  
11 | }
```

So again, there is no obvious "best" style. Just being consistent.

## 6 - DRY Principle

DRY stands for Don't Repeat Yourself. Also known as DIE: Duplication is Evil.

The principle states:

*"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."*

The purpose for most applications (or computers in general) is to automate repetitive tasks. This principle should be maintained in all code, even web applications. The same piece of code should not be repeated over and over again.

For example, most web applications consist of many pages. It's highly likely that these pages will contain common elements. Headers and footers are usually best candidates for this. It's not a good idea to keep copy pasting these headers and footers into every

page. [Here](#) is Jeffrey Way explaining how to create templates in CodeIgniter.

```
1  $this->load->view('includes/header');
2
3  $this->load->view($main_content);
4
5  $this->load->view('includes/footer');
```

## 7 - Avoid Deep Nesting

Too many levels of nesting can make code harder to read and follow.

```
01  function do_stuff() {
02
03  // ...
04
05      if (is_writable($folder)) {
06
07          if ($fp = fopen($file_path, 'w')) {
08
09              if ($stuff = get_some_stuff()) {
10
11                  if (fwrite($fp, $stuff)) {
12
13                      // ...
14
15                      } else {
16                          return false;
17                      }
18                  } else {
19                      return false;
20                  }
21              } else {
22                  return false;
23              }
24          } else {
25              return false;
26          }
27      }
```

For the sake of readability, it is usually possible to make changes to your code to reduce the level of nesting:

```
01  function do_stuff() {
02
03  // ...
```



```

04
05     if (!is_writable($folder)) {
06         return false;
07     }
08
09     if (!$fp = fopen($file_path, 'w')) {
10         return false;
11     }
12
13     if (!$stuff = get_some_stuff()) {
14         return false;
15     }
16
17     if (fwrite($fp, $stuff)) {
18         // ...
19     } else {
20         return false;
21     }
22 }

```

## 8 - Limit Line Length

Our eyes are more comfortable when reading tall and narrow columns of text. This is precisely the reason why newspaper articles look like this:



It is a good practice to avoid writing horizontally long lines of code.

```
01 // bad
02 $my_email->set_from('test@email.com')->add_to('programming@gmail.com')->set_subject('Me
03
04 // good
05 $my_email
06     ->set_from('test@email.com')
07     ->add_to('programming@gmail.com')
08     ->set_subject('Methods Chained')
09     ->set_body('Some long message')
10     ->send();
11
12 // bad
13 $query = "SELECT id, username, first_name, last_name, status FROM users LEFT JOIN user_
14
15 // good
16 $query = "SELECT id, username, first_name, last_name, status
17     FROM users
18     LEFT JOIN user_posts USING(users.id, user_posts.user_id)
19     WHERE post_id = '123'";
```

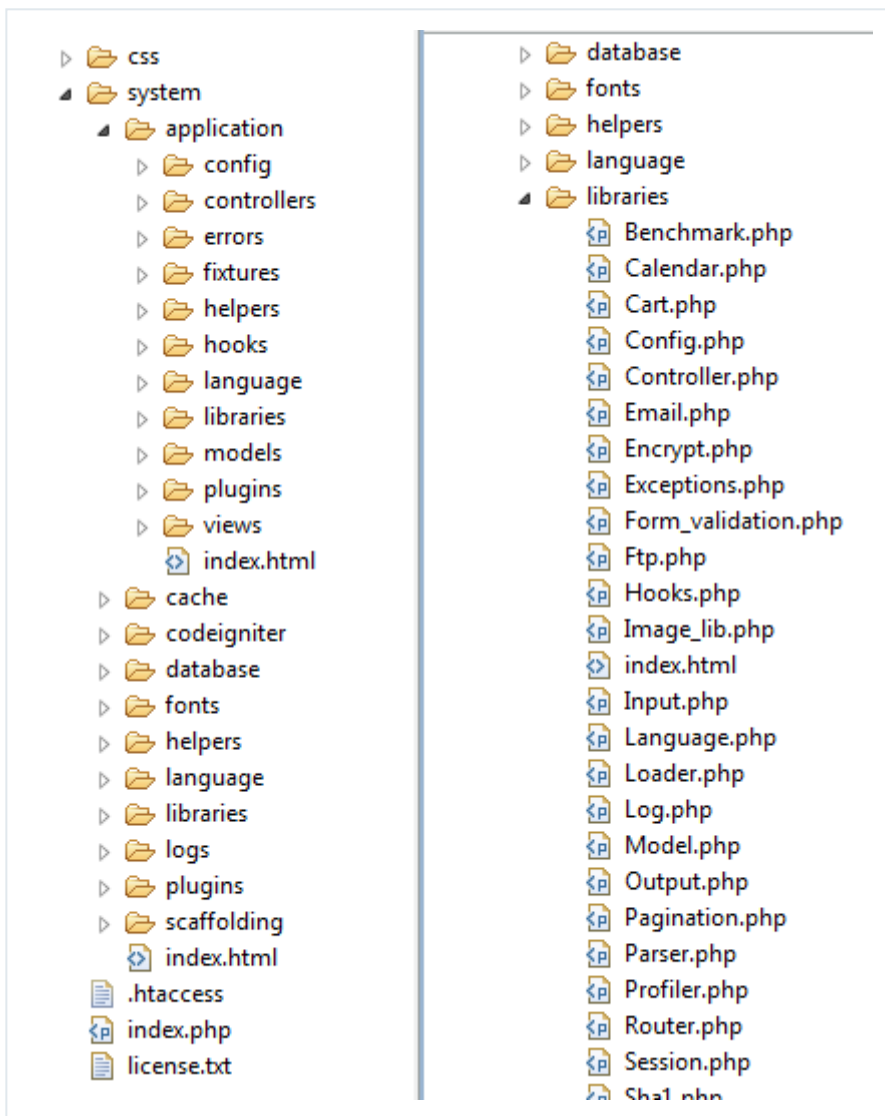
Also, if anyone intends to read the code from a terminal window, such as [Vim users](#), it is a good idea to limit the line length to around 80 characters.

## 9 - File and Folder Organization

Technically, you could write an entire application code within a single file. But that would prove to be a nightmare to read and maintain.

During my first programming projects, I knew about the idea of creating "include files." However, I was not yet even remotely organized. I created an "inc" folder, with two files in it: `db.php` and `functions.php`. As the applications grew, the functions file also became huge and unmaintainable.

One of the best approaches is to either [use a framework](#), or imitate their folder structure. Here is what CodeIgniter looks like:



## 10 - Consistent Temporary Names

Normally, the variables should be descriptive and contain one or more words. But, this doesn't necessarily apply to temporary variables. They can be as short as a single character.

It is a good practice to use consistent names for your temporary variables that have the same kind of role. Here are a few examples that I tend use in my code:

```
01 // $i for loop counters
02 for ($i = 0; $i < 100; $i++) {
03
04     // $j for the nested loop counters
05     for ($j = 0; $j < 100; $j++) {
```

```

06     }
07 }
08
09
10 // $ret for return variables
11 function foo() {
12     $ret['bar'] = get_bar();
13     $ret['stuff'] = get_stuff();
14
15     return $ret;
16 }
17
18 // $k and $v in foreach
19 foreach ($some_array as $k => $v) {
20
21 }
22
23 // $q, $r and $d for mysql
24 $q = "SELECT * FROM table";
25 $r = mysql_query($q);
26 while ($d = mysql_fetch_assoc($r)) {
27
28 }
29
30 // $fp for file pointers
31 $fp = fopen('file.txt', 'w');

```

## 11 - Capitalize SQL Special Words

Database interaction is a big part of most web applications. If you are writing raw SQL queries, it is a good idea to keep them readable as well.

Even though SQL special words and function names are case insensitive, it is common practice to capitalize them to distinguish them from your table and column names.

```

01 SELECT id, username FROM user;
02
03 UPDATE user SET last_login = NOW()
04 WHERE id = '123'
05
06 SELECT id, username FROM user u
07 LEFT JOIN user_address ua ON(u.id = ua.user_id)
08 WHERE ua.state = 'NY'
09 GROUP BY u.id
10 ORDER BY u.username
11 LIMIT 0,20

```

## 12 - Separation of Code and Data

This is another principle that applies to almost all programming languages in all environments. In the case of web development, the "data" usually implies HTML output.

When PHP was first released many years ago, it was primarily seen as a template engine. It was common to have big HTML files with a few lines of PHP code in between. However, things have changed over the years and websites became more and more dynamic and functional. The code is now a huge part of web applications, and it is no longer a good practice to combine it with the HTML.

You can either apply the principle to your application by yourself, or you can use a third party tool (template engines, frameworks or CMS's) and follow their conventions.

Popular PHP Frameworks:

- [CodeIgniter](#)
- [Zend Framework](#)
- [Cake PHP](#)
- [Symfony](#)

Popular Template Engines:

- [Smarty](#)
- [Dwoo](#)
- [Savant](#)

Popular Content Management Systems

## 13 - Alternate Syntax Inside Templates

You may choose not to use a fancy template engine, and instead go with plain inline PHP in your template files. This does not necessarily violate the "Separation of Code and Data," as long as the inline code is directly related to the output, and is readable. In this case you

should consider using the [alternate syntax](#) for control structures.

Here is an example:

```
01 <div class="user_controls">
02     <?php if ($user = Current_User::user()): ?>
03         Hello, <em><?php echo $user->username; ?></em> <br/>
04         <?php echo anchor('logout', 'Logout'); ?>
05     <?php else: ?>
06         <?php echo anchor('login', 'Login'); ?> |
07         <?php echo anchor('signup', 'Register'); ?>
08     <?php endif; ?>
09 </div>
10
11 <h1>My Message Board</h1>
12
13 <?php foreach($categories as $category): ?>
14
15     <div class="category">
16
17         <h2><?php echo $category->title; ?></h2>
18
19         <?php foreach($category->Forums as $forum): ?>
20
21             <div class="forum">
22
23                 <h3>
24                     <?php echo anchor('forums/'.$forum->id, $forum->title) ?>
25                     (<?php echo $forum->Threads->count(); ?> threads)
26                 </h3>
27
28                 <div class="description">
29                     <?php echo $forum->description; ?>
30                 </div>
31
32             </div>
33
34         <?php endforeach; ?>
35
36     </div>
37
38 <?php endforeach; ?>
```

This lets you avoid lots of curly braces. Also, the code looks and feels similar to the way HTML is structured and indented.

## 14 - Object Oriented vs. Procedural

Object oriented programming can help you create well structured code. But that does not mean you need to abandon procedural programming completely. Actually creating a mix of both styles can be good.

Objects should be used for representing data, usually residing in a database.

```
01  class User {
02
03      public $username;
04      public $first_name;
05      public $last_name;
06      public $email;
07
08      public function __construct() {
09          // ...
10      }
11
12      public function create() {
13          // ...
14      }
15
16      public function save() {
17          // ...
18      }
19
20      public function delete() {
21          // ...
22      }
23
24  }
```

Procedural functions may be used for specific tasks that can be performed independently.

```
1  function capitalize($string) {
2
3      $ret = strtoupper($string[0]);
4      $ret .= strtolower(substr($string,1));
5      return $ret;
6
7  }
```

## 15 - Read Open Source Code

Open Source projects are built with the input of many developers. These projects need to maintain a high level of code readability so that the team can work together as efficiently

as possible. Therefore, it is a good idea to browse through the source code of these projects to observe what these developers are doing.

descriptive  
file header

separator ->

class  
documentation

array elements  
tab indented

separator ->

class method

Exceptions.php

```
1 <?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');
2 /**
3  * CodeIgniter
4  *
5  * An open source application development framework for PHP 4.3.2 or newer
6  *
7  * @package      CodeIgniter
8  * @author        ExpressionEngine Dev Team
9  * @copyright      Copyright (c) 2008 - 2009, EllisLab, Inc.
10 * @license       http://codeigniter.com/user_guide/license.html
11 * @link          http://codeigniter.com
12 * @since         Version 1.0
13 * @filesource
14 */
15
16 // -----
17
18 /**
19  * Exceptions Class
20  *
21  * @package      CodeIgniter
22  * @subpackage    Libraries
23  * @category      Exceptions
24  * @author        ExpressionEngine Dev Team
25  * @link          http://codeigniter.com/user_guide/libraries/exceptions.html
26 */
27 class CI_Exceptions {
28     var $action;
29     var $severity;
30     var $message;
31     var $filename;
32     var $line;
33     var $ob_level;
34
35     var $levels = array(
36         E_ERROR           => 'Error',
37         E_WARNING         => 'Warning',
38         E_PARSE           => 'Parsing Error',
39         E_NOTICE          => 'Notice',
40         E_CORE_ERROR      => 'Core Error',
41         E_CORE_WARNING    => 'Core Warning',
42         E_COMPILE_ERROR   => 'Compile Error',
43         E_COMPILE_WARNING => 'Compile Warning',
44         E_USER_ERROR       => 'User Error',
45         E_USER_WARNING    => 'User Warning',
46         E_USER_NOTICE     => 'User Notice',
47         E_STRICT           => 'Runtime Notice'
48     );
49
50
51     /**
52      * Constructor
53      *
54      */
55     function CI_Exceptions()
56     {
57         $this->ob_level = ob_get_level();
58         // Note: Do not Log messages from this constructor.
59     }
60
61     // -----
62
63     /**
64      * Exception Logger
65      *
66      * This function logs PHP generated error messages
67      *
68      * @access private
69      * @param string $error The error message
```



## Class method documentation

```
69 * @param string the error severity
70 * @param string the error string
71 * @param string the error filepath
72 * @param string the error line number
73 * @return string
74 */
75 function log_exception($severity, $message, $filepath, $line)
76 {
```

# 16 - Code Refactoring

When you "refactor," you make changes to the code without changing any of its functionality. You can think of it like a "clean up," for the sake of improving readability and quality.

This doesn't include bug fixes or the addition of any new functionality. You might refactor code that you have written the day before, while it's still fresh in your head, so that it is more readable and reusable when you may potentially look at it two months from now. As the motto says: "refactor early, refactor often."

You may apply any of the "best practices" of code readability during the refactoring process.

I hope you enjoyed this article! Any that I missed? Let me know via the comments. And if you want to improve your coding, there are lots of scripts and apps available to help you on [Envato Market](#). See what's [most popular this week](#).



## Burak Guzel

Burak Guzel is a full time PHP Web Developer living in Arizona, originally from Istanbul, Turkey. He has a bachelors degree in Computer Science and Engineering from The Ohio State University. He has over 8 years of experience with PHP and MySQL. You can read more of his articles on his website at [PHPandStuff.com](http://PHPandStuff.com) and follow him on Twitter [here](#).

---



tuts+

**STUDENT ACCESS**  
JUST \$90/YR

Courses, eBooks & more >