Coding Standards

Table of Contents

Note: The PEAR Coding Standards apply to code that is part of the official PEAR distribution. Coding standards often abbreviated as CS among developers and they aim to keep code consistent to be easily readable and maintainable by most of PEAR folks.

The original coding standards have been adjusted several times through RFCs:

- Coding standard enhancements
- Error Handling Guidelines for PHP5 packages
- Header Comment Blocks
- Protected members
- Requiring E_STRICT Compatibility for New PEAR Packages

Those RFCs have been integrated into this document.

Note: The PEAR2 Coding Standards define several other rules that have to be followed once PEAR2 is in place.

Indenting and Line Length

Use an indent of 4 spaces, with no tabs. This helps to avoid problems with diffs, patches, SVN history and annotations.

For Emacs you should set indent-tabs-mode to nil. Here is an example mode hook that will set up Emacs (ensure that it is called when you are editing PHP files):

```
(defun pear/php-mode-init()
  "Set some buffer-local variables."
  (setq case-fold-search t)
  (c-set-offset 'arglist-intro '+)
  (c-set-offset 'arglist-close '0)
)
(add-hook 'php-mode-hook 'pear/php-mode-init)
```

Here are Vim rules for the same thing:

```
set expandtab
set shiftwidth=4
set softtabstop=4
set tabstop=4
```

It is recommended to keep lines at approximately 75-85 characters long for better code readability. Paul M. Jones has some thoughts about that limit.

Control Structures

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

```
<?php
if ((condition1) || (condition2)) {
    action1;
} elseif ((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.

For switch statements:

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

case 2:
    action2;
    break;

default:
    defaultaction;
    break;
}
?>
```

Split long if statements onto several lines

Long if statements may be split onto several lines when the character/line limit would be exceeded. The conditions have to be positioned onto the following line, and indented 4 characters. The logical operators (`&&`, `||`, etc.) should be at the beginning of the line to make it easier to comment (and exclude) the condition. The closing parenthesis and opening brace get their own line at the end of the conditions.

Keeping the operators at the beginning of the line has two advantages: It is trivial to comment out a particular line during development while keeping syntactically correct code (except of course the first line). Further is the logic kept at the front where it's not forgotten. Scanning such conditions is very easy since they are aligned below each other.

```
<?php

if (($condition1
    || $condition2)
    && $condition3
    && $condition4
) {
    //code here
}
?>
```

The first condition may be aligned to the others.

```
<?php

if (   $condition1
    || $condition2
    || $condition3
) {
    //code here
}
?>
```

The best case is of course when the line does not need to be split. When the if clause is really long enough to be split, it might be better to simplify it. In such cases, you could express conditions as variables an compare them in the `if()` condition. This has the benefit of "naming" and splitting the condition sets into smaller, better understandable chunks:

```
<?php

$is_foo = ($condition1 || $condition2);
$is_bar = ($condition3 && $condtion4);
if ($is_foo && $is_bar) {
    // ....
}
?>
```

> **Note:** There were suggestions to indent the parantheses "groups" by 1 space for each grouping. This is too hard to achieve in your coding flow, since your tab key always produces 4 spaces. Indenting the if clauses would take too much finetuning.

Ternary operators

The same rule as for if clauses also applies for the ternary operator: It may be split onto several lines, keeping the question mark and the colon at the front.

```
<?php

$a = $condition1 && $condition2
    ? $foo : $bar;

$b = $condition3 && $condition4
    ? $foo_man_this_is_too_long_what_should_i_do
    : $bar;
?>
```

Prev Coding Standards Next
Indenting and Line Length PEAR Manual Function Calls

Function Calls
Prev Next

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:

```
<?php
$var = foo($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:

```
<?php
$short         = foo($bar);
$long_variable = foo($baz);
?>
```

To support readability, parameters in subsequent calls to the same function/method may be aligned by parameter name:

```
<?php

$this->callSomeFunction('param1',     'second',        true);
$this->callSomeFunction('parameter2', 'third',         false);
$this->callSomeFunction('3',          'verrrrrrylong', true);
?>
```

Split function call on several lines

The CS require lines to have a maximum length of 80 chars. Calling functions or methods with many parameters while adhering to CS is impossible in that cases. It is allowed to split parameters in function calls onto several lines.

```
<?php

$this->someObject->subObject->callThisFunctionWithALongName(
    $parameterOne, $parameterTwo,
    $aVeryLongParameterThree
);
?>
```

Several parameters per line are allowed. Parameters need to be indented 4 spaces compared to the level of the function call. The opening parenthesis is to be put at the end of the function call line, the closing parenthesis gets its own line at the end of the parameters. This shows a visual end to the parameter indentations and follows the opening/closing brace rules for functions and conditionals.

The same applies not only for parameter variables, but also for nested function calls and for arrays.

```
<?php

$this->someObject->subObject->callThisFunctionWithALongName(
    $this->someOtherFunc(
        $this->someEvenOtherFunc(
            'Help me!',
            array(
                'foo'  => 'bar',
                'spam' => 'eggs',
            ),
            23
        ),
        $this->someEvenOtherFunc()
    ),
    $this->wowowowowow(12)
);
?>
```

Nesting those function parameters is allowed if it helps to make the code more readable, not only when it is necessary when the characters per line limit is reached.

Using fluent application programming interfaces often leads to many concatenated function calls. Those calls may be split onto several lines. When doing this, all subsequent lines are indented by 4 spaces and begin with the "->" arrow.

```
<?php

$someObject->someFunction("some", "parameter")
```

```
    ->someOtherFunc(23, 42)
    ->andAThirdFunction();
?>
```

Alignment of assignments

To support readability, the equal signs may be aligned in block-related assignments:

```
<?php

$short  = foo($bar);
$longer = foo($baz);
?>
```

The rule can be broken when the length of the variable name is at least 8 characters longer/shorter than the previous one:

```
<?php

$short = foo($bar);
$thisVariableNameIsVeeeeeeeeeeryLong = foo($baz);
?>
```

Split long assigments onto several lines

Assigments may be split onto several lines when the character/line limit would be exceeded. The equal sign has to be positioned onto the following line, and indented by 4 characters.

```
<?php

$GLOBALS['TSFE']->additionalHeaderData[$this->strApplicationName]
    = $this->xajax->getJavascript(t3lib_extMgm::siteRelPath('nr_xajax'));
?>
```

Class Definitions

Class declarations have their opening brace on a new line:

```
<?php
class Foo_Bar
{

    //... code goes here

}
?>
```

Function Definitions

Function declarations follow the "K&R style":

```
<?php
function fooFunction($arg1, $arg2 = '')
{
    if (condition) {
        statement;
    }
    return $val;
}
?>
```

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. Here is a slightly longer example:

```
<?php
function connect(&$dsn, $persistent = false)
{
    if (is_array($dsn)) {
        $dsninfo = &$dsn;
    } else {
        $dsninfo = DB::parseDSN($dsn);
    }

    if (!$dsninfo || !$dsninfo['phptype']) {
        return $this->raiseError();
    }
    return true;
}
?>
```

Split function definitions onto several lines

Functions with many parameters may need to be split onto several lines to keep the 80 characters/line limit. The first parameters may be put onto the same line as the function name if there is enough space. Subsequent parameters on following lines are to be indented 4 spaces. The closing parenthesis and the opening brace are to be put onto the next line, on the same indentation level as the "`function`" keyword.

```
<?php

function someFunctionWithAVeryLongName($firstParameter = 'something', $secondParameter = 'booooo',
    $third = null, $fourthParameter = false, $fifthParameter = 123.12,
    $sixthParam = true
) {
    //....
?>
```

Prev Coding Standards Next
Class Definitions PEAR Manual Arrays

Arrays
Prev Next

Arrays

Assignments in arrays may be aligned. When splitting array definitions onto several lines, the last value may also have a trailing comma. This is valid PHP syntax and helps to keep code diffs minimal:

```
<?php

$some_array = array(
    'foo'  => 'bar',
    'spam' => 'ham',
);
?>
```

Prev Coding Standards Next
Function Definitions PEAR Manual Comments

Comments
Prev Next

Comments

Complete inline documentation comment blocks (docblocks) must be provided. Please read the Sample File and Header Comment Blocks sections of the Coding Standards to learn the specifics of writing docblocks for PEAR packages. Further information can be found on the phpDocumentor website.

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.

C style comments (/* */) and standard C++ comments (//) are both fine. Use of Perl/shell style comments (#) is discouraged.

Prev Coding Standards Next
Arrays PEAR Manual Including Code

Including Code
Prev Next

Including Code

Anywhere you are unconditionally including a class file, use **require_once**. Anywhere you are conditionally including a class file (for example, factory methods), use **include_once**. Either of these will ensure that class files are included only once. They share the same file list, so you don't need to worry about mixing them - a file included with **require_once** will not be included again by **include_once**.

> **Note: include_once** and **require_once** are statements, not functions. Parentheses should not surround the subject filename.

Prev Coding Standards Next
Comments PEAR Manual PHP Code Tags

PHP Code Tags
Prev Next

PHP Code Tags

Always use `<?php ?>` to delimit PHP code, not the `<? ?>` shorthand. This is required for PEAR compliance and is also the most portable way to include PHP code on differing operating systems and setups.

Header Comment Blocks

All source code files in the PEAR repository shall contain a "page-level" docblock at the top of each file and a "class-level" docblock immediately above each class. Below are examples of such docblocks.

```
<?php

/* vim: set expandtab tabstop=4 shiftwidth=4 softtabstop=4: */

/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * PHP version 5
 *
 * LICENSE: This source file is subject to version 3.01 of the PHP license
 * that is available through the world-wide-web at the following URI:
 * http://www.php.net/license/3_01.txt.  If you did not receive a copy of
 * the PHP License and are unable to obtain it through the web, please
 * send a note to license@php.net so we can mail you a copy immediately.
 *
 * @category   CategoryName
 * @package    PackageName
 * @author     Original Author <author@example.com>
 * @author     Another Author <another@example.com>
 * @copyright  1997-2005 The PHP Group
 * @license    http://www.php.net/license/3_01.txt  PHP License 3.01
 * @version    SVN: $Id$
 * @link       http://pear.php.net/package/PackageName
 * @see        NetOther, Net_Sample::Net_Sample()
 * @since      File available since Release 1.2.0
 * @deprecated File deprecated in Release 2.0.0
 */

/*
* Place includes, constant defines and $_GLOBAL settings here.
* Make sure they have appropriate docblocks to avoid phpDocumentor
* construing they are documented by the page-level docblock.
*/

/**
 * Short description for class
 *
 * Long description for class (if any)...
 *
 * @category   CategoryName
 * @package    PackageName
 * @author     Original Author <author@example.com>
 * @author     Another Author <another@example.com>
 * @copyright  1997-2005 The PHP Group
 * @license    http://www.php.net/license/3_01.txt  PHP License 3.01
 * @version    Release: @package_version@
 * @link       http://pear.php.net/package/PackageName
 * @see        NetOther, Net_Sample::Net_Sample()
 * @since      Class available since Release 1.2.0
 * @deprecated Class deprecated in Release 2.0.0
 */
class Foo_Bar
{
}

?>
```

Required Tags That Have Variable Content

Short Descriptions

Short descriptions must be provided for all docblocks. They should be a quick sentence, not the name of the item. Please read the Coding Standard's Sample File about how to write good descriptions.

PHP Versions

One of the following must go in the page-level docblock:

```
* PHP version 4
 * PHP version 5
 * PHP versions 4 and 5
```

@license

There are several possible licenses. One of the following must be picked and placed in the page-level and class-level docblocks:

```
* @license   http://www.apache.org/licenses/LICENSE-2.0  Apache License 2.0
 * @license   http://www.freebsd.org/copyright/freebsd-license.html  BSD License (2 Clause)
 * @license   http://www.debian.org/misc/bsd.license  BSD License (3 Clause)
 * @license   http://www.freebsd.org/copyright/license.html  BSD License (4 Clause)
 * @license   http://www.opensource.org/licenses/mit-license.html  MIT License
 * @license   http://www.gnu.org/copyleft/lesser.html  LGPL License 2.1
 * @license   http://www.php.net/license/3_01.txt  PHP License 3.01
```

For more information, see the PEAR Group's Licensing Announcement.

@link

The following must be used in both the page-level and class-level docblocks. Of course, change "PackageName" to the name of your package. This ensures the generated documentation links back your package.

```
* @link      http://pear.php.net/package/PackageName
```

@author

There's no hard rule to determine when a new code contributor should be added to the list of authors for a given source file. In general, their changes should fall into the "substantial" category (meaning somewhere around 10% to 20% of code changes). Exceptions could be made for rewriting functions or contributing new logic.

Simple code reorganization or bug fixes would not justify the addition of a new individual to the list of authors.

@since

This tag is required when a file or class is added after the package's initial release. Do not use it in an initial release.

@deprecated

This tag is required when a file or class is no longer used but has been left in place for backwards compatibility.

Optional Tags

@copyright

Feel free to apply whatever copyrights you desire. When formatting this tag, the year should be in four digit format and if a span of years is involved, use a hyphen between the earliest and latest year. The copyright holder can be you, a list of people, a company, the PHP Group, etc. Examples:

```
* @copyright 2003 John Doe and Jennifer Buck
 * @copyright 2001-2004 John Doe
 * @copyright 1997-2004 The PHP Group
 * @copyright 2001-2004 XYZ Corporation
```

License Summary

If you are using the PHP License, use the summary text provided above. If another license is being used, please remove the PHP License summary. Feel free to substitute it with text appropriate to your license, though to keep things easy to locate, please preface the text with `LICENSE:` .

@see

Add a @see tag when you want to refer users to other sections of the package's documentation. If you have multiple items, separate them with commas rather than adding multiple @see tags.

Order and Spacing

To ease long term readability of PEAR source code, the text and tags must conform to the order and spacing provided in the example above. This standard is adopted from the JavaDoc standard.

@package_version@ Usage

There are two ways to implement the @package_version@ replacements. The procedure depends on whether you write your own package.xml files or if you use the PackageFileManager.

For those authoring package.xml files directly, add a <replace> element for each file. The XML for such would look something like this:

```
<file name="Class.php">
  <replace from="@package_version@" to="version" type="package-info" />
</file>
```

Maintainers using the PackageFileManager need to call **addReplacement()** for each file:

```
<?php
$pkg->addReplacement('filename.php', 'package-info',
                     '@package_version@', 'version');
?>
```

Transition Policy

Existing Small Packages

Existing packages that have only a few files are required to adopt these docblocks before the next release.

Existing Large Packages

> Existing packages with many files are encouraged to adopt the new headers as soon as possible. When such packages come out with a new major version upgrade, these docblocks must be implemented therein.

New and Unreleased Packages

> New packages and existing packages which have no releases yet must include these docblocks before their first release.

Prev Coding Standards Next
PHP Code Tags PEAR Manual Using SVN
Using SVN
Prev Next

Using SVN

This section applies only to packages using SVN at svn.php.net.

Include the Id SVN keyword in each file.

The rest of this section assumes that you have basic knowledge about SVN tags and branches.

SVN tags are used to label which revisions of the files in your package belong to a given release. Below is a list of the required and suggested SVN tags:

RELEASE_*n_n_n*

> (required) Used for tagging a release. If you don't use it, there's no way to go back and retrieve your package from the SVN server in the state it was in at the time of the release.

QA_*n_n_n*

> (branch, optional) If you feel you need to roll out a release candidate before releasing, it's a good idea to make a branch for it so you can isolate the release and apply only those critical fixes before the actual release. Meanwhile, normal development may continue on the main trunk.

MAINT_*n_n_n*

> (branch, optional) If you need to make "micro-releases" (for example 1.2.1 and so on after 1.2.0), you can use a branch for that too, if your main trunk is very active and you want only minor changes between your micro-releases.

Only the RELEASE tag is required, the rest are recommended for your convenience.

Below is an example of how to tag the 1.2.0 release of the **Money_Fast** package:

```
    $ SVN copy http://svn.php.net/repository/pear/packages/Money_Fast/trunk  http://svn.php.net/repository/pear/packages/Money_Fast/tags/RELEASE_1_2_0
```

By doing this you make it possible for the PEAR web site to take you through the rest of your release process.

Here's an example of how to create a QA branch:

```
$ svn copy http://svn.php.net/repository/pear/packages/Money_Fast/trunk http://svn.php.net/repository/pear/packages/Money_Fast/branches/QA_2_0_0

...

$ svn copy http://svn.php.net/repository/pear/packages/Money_Fast/branches/QA_2_0_0 http://svn.php.net/repository/pear/packages/Money_Fast/tags/RELEASE_2_0

...and then the actual release, from the same branch:

$ svn copy http://svn.php.net/repository/pear/packages/Money_Fast/branches/QA_2_0_0 http://svn.php.net/repository/pear/packages/Money_Fast/tags/RELEASE_2_0
```

Prev Coding Standards Next
Header Comment Blocks PEAR Manual Example URLs
Example URLs
Prev Next

Example URLs

Use `example.com`, `example.org` and `example.net` for all example URLs and email addresses, per RFC 2606.

Prev Coding Standards Next
Using SVN PEAR Manual Naming Conventions
Naming Conventions
Prev Next

Naming Conventions

Global Variables and Functions

If your package needs to define global variables, their names should start with a single underscore followed by the package name and another underscore. For example, the PEAR package uses a global variable called $_PEAR_destructor_object_list.

Global functions should be named using the "studly caps" style (also referred to as "bumpy case" or "camel caps"). In addition, they should have the package name as a prefix, to avoid name collisions between packages. The initial letter of the name (after the prefix) is lowercase, and each letter that starts a new "word" is capitalized. An example:

XML_RPC_serializeData()

Classes

Classes should be given descriptive names. Avoid using abbreviations where possible. Class names should always begin with an uppercase letter. The PEAR class hierarchy is also reflected in the class name, each level of the hierarchy separated with a single underscore. Examples of good class names are:

Log	Net_Finger	HTML_Upload_Error

Class Variables and Methods

Class variables (a.k.a properties) and methods should be named using the "studly caps" style (also referred to as "bumpy case" or "camel caps"). Some examples (these would be "public" members):

$counter	connect()	getData()	buildSomeWidget()

Private class members are preceded by a single underscore. For example:

$_status	_sort()	_initTree()

Note: The following applies to PHP5.

Protected class members are not preceded by a single underscore. For example:

protected $somevar	protected function initTree()

Constants

Constants should always be all-uppercase, with underscores to separate words. Prefix constant names with the uppercased name of the class/package they are used in. Some examples:

DB_DATASOURCENAME	SERVICES_AMAZON_S3_LICENSEKEY

Note: The `true`, `false` and `null` constants are excepted from the all-uppercase rule, and must always be lowercase.

Prev Coding Standards Next
Example URLs PEAR Manual File Formats

File Formats
Prev Next

File Formats

All scripts contributed to PEAR must:

- Be stored as ASCII text
- Use ISO-8859-1 or UTF-8 character encoding. The encoding may be declared using `declare(encoding = 'utf-8');` at the top of the file.
- Be Unix formatted

  "Unix formatted" means two things:

  1) Lines must end only with a line feed (`LF`). Line feeds are represented as ordinal `10`, octal `012` and hex `0A`. Do not use carriage returns (`CR`) like Macintosh computers do or the carriage return/line feed combination (`CRLF`) like Windows computers do.

  2) There should be *one* line feed after the closing PHP tag (`?>`). This means that when the cursor is at the very end of the file, it should be *one* line below the closing PHP tag.

Prev Coding Standards Next
Naming Conventions PEAR Manual E_STRICT-compatible code

E_STRICT-compatible code
Prev Next

`E_STRICT`-compatible code

Starting on 01 January 2007, all new code that is suggested for inclusion into PEAR must be `E_STRICT`-compatible. This means that it must not produce any warnings or errors when PHP's error reporting level is set to `E_ALL | E_STRICT`.

The development of existing packages that are not `E_STRICT`-compatible can continue as usual. If however a new *major* version of the package is released, this major version must then be `E_STRICT`-compatible.

More details on this part of the Coding Standards can be found in the corresponding RFC.

Prev Coding Standards Next
File Formats PEAR Manual Error Handling Guidelines
Error Handling Guidelines
Prev Next

Error Handling Guidelines

This part of the Coding Standards describes how errors are handled in PEAR packages that are developed for PHP 5 and 6. It uses Exceptions, introduced in PHP 5.0 with Zend Engine 2, as the error handling mechanism.

Definition of an error

An error is defined as an unexpected, invalid program state from which it is impossible to recover. For the sake of definition, recovery scope is defined as the method scope. Incomplete recovery is considered a recovery.

One pretty straightforward example for an error

```php
<?php
/*
 * Connect to Specified Database
 *
 * @throws Example_Datasource_Exception when it can't connect
 * to specified DSN.
 */
function connectDB($dsn)
{
    $this->db =& DB::connect($dsn);
    if (DB::isError($this->db)) {
        throw new Example_Datasource_Exception(
                "Unable to connect to $dsn:" . $this->db->getMessage()
        );
    }
}
?>
```

In this example the objective of the method is to connect to the given DSN. Since it can't do anything but ask PEAR DB to do it, whenever DB returns an error, the only option is to bail out and launch the exception.

Error handling with recovery

```php
<?php
/*
 * Connect to one of the possible databases
 *
 * @throws Example_Datasource_Exception when it can't connect to
 * any of the configured databases.
 *
 * @throws Example_Config_Exception when it can't find databases
 * in the configuration.
 */

function connect(Config $conf)
{
    $dsns =& $conf->searchPath(array('config', 'db'));
    if ($dsns === FALSE) throw new Example_Config_Exception(
        'Unable to find config/db section in configuration.'
    );

    $dsns =& $dsns->toArray();

    foreach($dsns as $dsn) {
        try {
            $this->connectDB($dsn);
            return;
        } catch (Example_Datasource_Exception e) {
            // Some warning/logging code recording the failure
            // to connect to one of the databases
        }
    }
    throw new Example_Datasource_Exception(
        'Unable to connect to any of the configured databases'
    );
}
?>
```

This second example shows an exception being caught and recovered from. Although the lower level **connectDB()** method is unable to do anything but throw an error when one database connection fails, the upper level **connect()** method knows the object can go by with any one of the configured databases. Since the error was recovered from, the exception is silenced at this level and not rethrown.

Incomplete recovery

```php
<?php
/*
 * loadConfig parses the provided configuration. If the configuration
 * is invalid, it will set the configuration to the default config.
 *
 */
function loadConfig(Config $conf)
{
    try {
        $this->config = $conf->parse();
    } catch (Config_Parse_Exception e) {
        // Warn/Log code goes here
        // Perform incomplete recovery
        $this->config = $this->defaultConfig;
    }
}
?>
```

The recovery produces side effects, so it is considered incomplete. However, the program may proceed, so the exception is considered handled, and must not be rethrown. As in the previous example, when silencing the exception, logging or warning should occur.

Error Signaling in PHP 5 PEAR packages

Error conditions in PEAR packages written for PHP 5 must be signaled using exceptions. Usage of return codes or return **PEAR_Error** objects is deprecated in favor of exceptions. Naturally, packages providing compatibility with PHP 4 do not fall under these coding guidelines, and may thus use the error handling mechanisms defined in the PHP 4 PEAR coding guidelines.

An exception should be thrown whenever an error condition is met, according to the definition provided in the previous section. The thrown exception should contain enough information to debug the error and quickly identify the error cause. Note that, during production runs, no exception should reach the end-user, so there is no need for concern about technical complexity in the exception error messages.

The basic **PEAR_Exception** contains a textual error, describing the program state that led to the throw and, optionally, a wrapped lower level exception, containing more info on the lower level causes of the error.

The kind of information to be included in the exception is dependent on the error condition. From the point of view of exception throwing, there are three classes of error conditions:

1. Errors detected during precondition checks
2. Lower level library errors signaled via error return codes or error return objects
3. Uncorrectable lower library exceptions

Errors detected during precondition checks should contain a description of the failed check. If possible, the description should contain the violating value. Naturally, no wrapped exception can be included, as there isn't a lower level cause of the error. Example:

```php
<?php
function divide($x, $y)
{
    if ($y == 0) {
        throw new Example_Aritmetic_Exception('Division by zero');
    }
}
?>
```

Errors signaled via return codes by lower level libraries, if unrecoverable, should be turned into exceptions. The error description should try to convey all information contained in the original error. One example, is the connect method previously presented:

```php
<?php
/*
 * Connect to Specified Database
 *
 * @throws Example_Datasource_Exception when it can't connect to specified DSN.
 */
function connectDB($dsn)
{
    $this->db =& DB::connect($dsn);
    if (DB::isError($this->db)) {
        throw new Example_Datasource_Exception(
                "Unable to connect to $dsn:" . $this->db->getMessage()
        );
    }
}
?>
```

Lower library exceptions, if they can't be corrected, should either be rethrown or bubbled up. When rethrowing, the original exception must be wrapped inside the one being thrown. When letting the exception bubble up, the exception just isn't handled and will continue up the call stack in search of a handler.

Rethrowing an exception

```php
<?php
function preTaxPrice($retailPrice, $taxRate)
{
    try {
        return $this->divide($retailPrice, 1 + $taxRate);
    } catch (Example_Aritmetic_Exception e) {
        throw new Example_Tax_Exception('Invalid tax rate.', e);
    }
}
?>
```

Letting exceptions bubble up

```
<?php
function preTaxPrice($retailPrice, $taxRate)
{
    return $this->divide($retailPrice, 1 + $taxRate);
}
?>
```

The case between rethrowing or bubbling up is one of software architecture: Exceptions should be bubbled up, except in these two cases:

1. The original exception is from another package. Letting it bubble up would cause implementation details to be exposed, violating layer abstraction, conducing to poor design.

2. The current method can add useful debugging information to the received error before rethrowing.

Exceptions and normal program flow

Exceptions should never be used as normal program flow. If removing all exception handling logic (try-catch statements) from the program, the remaining code should represent the "One True Path" -- the flow that would be executed in the absence of errors.

This requirement is equivalent to requiring that exceptions be thrown only on error conditions, and never in normal program states.

One example of a method that wrongly uses the bubble up capability of exceptions to return a result from a deep recursion:

```
<?php
/**
 * Recursively search a tree for string.
 * @throws ResultException
 */
public function search(TreeNode $node, $data)
{
    if ($node->data === $data) {
        throw new ResultException( $node );
    } else {
        search( $node->leftChild, $data );
        search( $node->rightChild, $data );
    }
}
?>
```

In the example the ResultException is simply using the "eject!" qualities of exception handling to jump out of deeply nested recursion. When actually used to signify an error this is a very powerful feature, but in the example above this is simply lazy development.

Exception class hierarchies

All of PEAR packages exceptions must be descendant from **PEAR_Exception**. **PEAR_Exception** provides exception wrapping abilities, absent from the top level PHP Exception class, and needed to comply with the previous section requirements.

Additionally, each PEAR package must provide a top level exception, named <Package_Name>_Exception. It is considered best practice that the package never throws exceptions that aren't descendant from its top level exception.

Documenting Exceptions

Because PHP, unlike Java, does not require you to explicitly state which exceptions a method throws in the method signature, it is critical that exceptions be thoroughly documented in your method headers.

Exceptions should be documented using the `@throws` phpdoc keyword

```
<?php
/**
 * This method searches for aliens.
 *
 * @return array Array of Aliens objects.
 * @throws AntennaBrokenException If the impedence readings indicate
 * that the antenna is broken.
 *
 * @throws AntennaInUseException If another process is using the
 * antenna already.
 */
public function findAliens($color = 'green');
?>
```

In many cases middle layers of an application will rewrap any lower-level exceptions into more meaningful application exceptions. This also needs to be made clear:

```
<?php
/**
 * Load session objects into shared memory.
 *
 * @throws LoadingException Any lower-level IOException will be wrapped
 * and re-thrown as a LoadingException.
 */
public function loadSessionObjects();
?>
```

In other cases your method may simply be a conduit through which lower level exceptions can pass freely. As challenging as it may be, your method should also document which exceptions it is *not* catching.

```
<?php
```

```
/**
 * Performs a batch of database queries (atomically, not in transaction).
 * @throws SQLException Low-level SQL errors will bubble up through this method.
 */
public function batchExecute();
?>
```

Exceptions as part of the API

Exceptions play a critical role in the API of your library. Developers using your library *depend* on accurate descriptions of where and why exceptions might be thrown from your package. Documentation is critical. Also maintaining the types of messages that are thrown is also an important requirement for maintaining backwards-compatibility.

Because Exceptions are critical to the API of your package, you must ensure that you don't break backwards compatibility (BC) by making changes to exceptions.

Things that break BC include:

- Any change to which methods throw exceptions.
- A change whereby a method throws an exception higher in the inheritance tree. For example, if you changed your method to throw a **PEAR_Exception** rather than a **PEAR_IOException**, you would be breaking backwards compatibility.

Things that do not break BC:

- Throwing a subclass of the original exception. For example, changing a method to throw **PEAR_IOException** when before it had been throwing **PEAR_Exception** would not break BC (provided that **PEAR_IOException** extends **PEAR_Exception**).

Prev | Coding Standards | Next
E_STRICT-compatible code | PEAR Manual | Best practices
Best practices
Prev | | Next

Best practices

There are other things not covered by PEAR Coding Standards which are mostly subject of personal preference and not directly related to readability of the code. Things like "single quotes vs double quotes" are features of PHP itself to make programming easier and there are no reasons not use one way in preference to another. Such best practices are left solely on developer to decide. The only recommendation could be made to keep consistency within package and respect personal style of other developers.

Readability of code blocks

Related lines of code should be grouped into blocks, seperated from each other to keep readability as high as possible. The definition of "related" depends on the code :)

For example:

```
<?php

if ($foo) {
    $bar = 1;
}
if ($spam) {
    $ham = 1;
}
if ($pinky) {
    $brain = 1;
}
?>
```

is a lot easier to read when seperated:

```
<?php

if ($foo) {
    $bar = 1;
}

if ($spam) {
    $ham = 1;
}

if ($pinky) {
    $brain = 1;
}
?>
```

Return early

To keep readability in functions and methods, it is wise to return early if simple conditions apply that can be checked at the beginning of a method:

```
<?php

function foo($bar, $baz)
{
    if ($foo) {
        //assume
        //that
        //here
        //is
        //the
        //whole
        //logic
        //of
        //this
        //method
        return $calculated_value;
    } else {
        return null;
    }
}
?>
```

It's better to return early, keeping indentation and brain power needed to follow the code low.

```
<?php

function foo($bar, $baz)
{
    if (!$foo) {
        return null;
    }

    //assume
    //that
    //here
    //is
    //the
    //whole
    //logic
    //of
    //this
    //method
    return $calculated_value;
}
?>
```

Prev Coding Standards Next
Error Handling Guidelines PEAR Manual Sample File (including Docblock Comment standards)

Sample File (including Docblock Comment standards)
Prev Next

Sample File (including Docblock Comment standards)

The source code of PEAR packages are read by thousands of people. Also, it is likely other people will become developers on your package at some point in the future. Therefore, it is important to make life easier for everyone by formatting the code and docblocks in standardized ways. People can then quickly find the information they are looking for because it is in the expected location. Your cooperation is appreciated.

Each docblock in the example contains many details about writing Docblock Comments. Following those instructions is important for two reasons. First, when docblocks are easy to read, users and developers can quickly ascertain what your code does. Second, the PEAR website now contains the phpDocumentor generated documentation for each release of each package, so keeping things straight here means the API docs on the website will be useful.

Please take note of the vertical and horizontal spacing. They are part of the standard.

The "fold markers" (`// {{{` and `// }}}`) are optional. If you aren't using fold markers, remove `foldmethod=marker` from the vim header.

```
<?php

/* vim: set expandtab tabstop=4 shiftwidth=4 softtabstop=4: */

/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * PHP version 5
 *
 * LICENSE: This source file is subject to version 3.01 of the PHP license
 * that is available through the world-wide-web at the following URI:
 * http://www.php.net/license/3_01.txt.  If you did not receive a copy of
 * the PHP License and are unable to obtain it through the web, please
 * send a note to license@php.net so we can mail you a copy immediately.
 *
 * @category   CategoryName
 * @package    PackageName
 * @author     Original Author <author@example.com>
 * @author     Another Author <another@example.com>
 * @copyright  1997-2005 The PHP Group
 * @license    http://www.php.net/license/3_01.txt  PHP License 3.01
 * @version    SVN: $Id$
 * @link       http://pear.php.net/package/PackageName
 * @see        NetOther, Net_Sample::Net_Sample()
 * @since      File available since Release 1.2.0
 * @deprecated File deprecated in Release 2.0.0
 */

/**
 * This is a "Docblock Comment," also known as a "docblock."  The class'
 * docblock, below, contains a complete description of how to write these.
 */
require_once 'PEAR.php';
```

```
// {{{ constants

/**
 * Methods return this if they succeed
 */
define('NET_SAMPLE_OK', 1);

// }}}
// {{{ GLOBALS

/**
 * The number of objects created
 * @global int $GLOBALS['_NET_SAMPLE_Count']
 */
$GLOBALS['_NET_SAMPLE_Count'] = 0;

// }}}
// {{{ Net_Sample

/**
 * An example of how to write code to PEAR's standards
 *
 * Docblock comments start with "/**" at the top.  Notice how the "/"
 * lines up with the normal indenting and the asterisks on subsequent rows
 * are in line with the first asterisk.  The last line of comment text
 * should be immediately followed on the next line by the closing asterisk
 * and slash and then the item you are commenting on should be on the next
 * line below that.  Don't add extra lines.  Please put a blank line
 * between paragraphs as well as between the end of the description and
 * the start of the @tags.  Wrap comments before 80 columns in order to
 * ease readability for a wide variety of users.
 *
 * Docblocks can only be used for programming constructs which allow them
 * (classes, properties, methods, defines, includes, globals).  See the
 * phpDocumentor documentation for more information.
 * http://phpdoc.org/docs/HTMLSmartyConverter/default/phpDocumentor/tutorial_phpDocumentor.howto.pkg.html
 *
 * The Javadoc Style Guide is an excellent resource for figuring out
 * how to say what needs to be said in docblock comments.  Much of what is
 * written here is a summary of what is found there, though there are some
 * cases where what's said here overrides what is said there.
 * http://java.sun.com/j2se/javadoc/writingdoccomments/index.html#styleguide
 *
 * The first line of any docblock is the summary.  Make them one short
 * sentence, without a period at the end.  Summaries for classes, properties
 * and constants should omit the subject and simply state the object,
 * because they are describing things rather than actions or behaviors.
 *
 * Below are the tags commonly used for classes. @category through @version
 * are required.  The remainder should only be used when necessary.
 * Please use them in the order they appear here.  phpDocumentor has
 * several other tags available, feel free to use them.
 *
 * @category   CategoryName
 * @package    PackageName
 * @author     Original Author <author@example.com>
 * @author     Another Author <another@example.com>
 * @copyright  1997-2005 The PHP Group
 * @license    http://www.php.net/license/3_01.txt  PHP License 3.01
 * @version    Release: @package_version@
 * @link       http://pear.php.net/package/PackageName
 * @see        NetOther, Net_Sample::Net_Sample()
 * @since      Class available since Release 1.2.0
 * @deprecated Class deprecated in Release 2.0.0
 */
class Net_Sample
{
    // {{{ properties

    /**
     * The status of foo's universe
     *
     * Potential values are 'good', 'fair', 'poor' and 'unknown'.
     *
     * @var string
     */
    var $foo = 'unknown';

    /**
     * The status of life
     *
     * Note that names of private properties or methods must be
     * preceeded by an underscore.
     *
     * @var bool
     * @access private
     */
    var $_good = true;

    // }}}
    // {{{ setFoo()

    /**
     * Registers the status of foo's universe
     *
     * Summaries for methods should use 3rd person declarative rather
     * than 2nd person imperative, beginning with a verb phrase.
     *
     * Summaries should add description beyond the method's name. The
     * best method names are "self-documenting", meaning they tell you
     * basically what the method does.  If the summary merely repeats
     * the method name in sentence form, it is not providing more
     * information.
     *
     * Summary Examples:
     *   + Sets the label              (preferred)
     *   + Set the label               (avoid)
     *   + This method sets the label  (avoid)
     *
     * Below are the tags commonly used for methods.  A @param tag is
     * required for each parameter the method has.  The @return
     * and @access tags are mandatory.  The @throws tag is required if
     * the method uses exceptions.  @static is required if the method can
     * be called statically.  The remainder should only be used when
     * necessary.  Please use them in the order they appear here.
     * phpDocumentor has several other tags available, feel free to use
     * them.
     *
     * The @param tag contains the data type, then the parameter's
     * name, followed by a description.  By convention, the first noun in
     * the description is the data type of the parameter.  Articles like
     * "a", "an", and  "the" can precede the noun.  The descriptions
     * should start with a phrase.  If further description is necessary,
     * follow with sentences.  Having two spaces between the name and the
     * description aids readability.
     *
     * When writing a phrase, do not capitalize and do not end with a
     * period:
     *   + the string to be tested
     *
     * When writing a phrase followed by a sentence, do not capitalize the
     * phrase, but end it with a period to distinguish it from the start
     * of the next sentence:
     *   + the string to be tested. Must use UTF-8 encoding.
     *
     * Return tags should contain the data type then a description of
     * the data returned.  The data type can be any of PHP's data types
     * (int, float, bool, string, array, object, resource, mixed)
     * and should contain the type primarily returned.  For example, if
     * a method returns an object when things work correctly but false
     * when an error happens, say 'object' rather than 'mixed.'  Use
     * 'void' if nothing is returned.
     *
     * Here's an example of how to format examples:
     * <code>
     * require_once 'Net/Sample.php';
     *
     * $s = new Net_Sample();
     * if (PEAR::isError($s)) {
     *     echo $s->getMessage() . "\n";
     * }
     * </code>
     *
     * Here is an example for non-php example or sample:
     * <samp>
     * pear install net_sample
     * </samp>
     *
     * @param string $arg1 the string to quote
     * @param int    $arg2 an integer of how many problems happened.
     *                     Indent to the description's starting point
     *                     for long ones.
     *
     * @return int the integer of the set mode used. FALSE if foo
     *             foo could not be set.
     * @throws exceptionclass [description]
     *
     * @access public
     * @static
     * @see Net_Sample::$foo, Net_Other::someMethod()
     * @since Method available since Release 1.2.0
     * @deprecated Method deprecated in Release 2.0.0
     */
    function setFoo($arg1, $arg2 = 0)
    {
        /*
         * This is a "Block Comment."  The format is the same as
         * Docblock Comments except there is only one asterisk at the
         * top.  phpDocumentor doesn't parse these.
         */
        if ($arg1 == 'good' || $arg1 == 'fair') {
            $this->foo = $arg1;
            return 1;
        } elseif ($arg1 == 'poor' && $arg2 > 1) {
            $this->foo = 'poor';
            return 2;
        } else {
            return false;
        }
    }

    // }}}
}

// }}}

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * c-hanging-comment-ender-p: nil
 * End:
 */

?>
```

Prev | Coding Standards | Next
Best practices | PEAR Manual | The PEAR toolbox

The PEAR toolbox
Prev | | Next

The PEAR toolbox

PEAR provides some tools to help developers keep their code clean and free of coding standards related errors.

For one there is PHP_CodeSniffer which can be used to detect coding standard errors in your scripts. Further, whole PEAR SVN repository is checked each night for violations - the results can be found at PEAR QA results overview page as well as on the linked subpages which explain the errors in detail.

Prev | Coding Standards | Next
Sample File (including Docblock Comment standards) | PEAR Manual | PEAR2 Coding Standards