TYPO3 v4 Coding Guidelines

Extension Key: doc_core_cgl
Language: en
Version: 4.5.1
Keywords: forDevelopers, forIntermediates

The content of this document is related to TYPO3
- a GNU/GPL CMS/Framework available from www.typo3.org

Official documentation

This document is included as part of the official TYPO3 documentation. It has been approved by the TYPO3 Documentation Team following a peer-review process. The reader should expect the information in this document to be accurate - please report discrepancies to the Documentation Team (documentation@typo3.org). Official documents are kept up-to-date to the best of the Documentation Team's abilities.

Core Manual

This document is a Core Manual. Core Manuals address the built in functionality of TYPO3 and are designed to provide the reader with in-depth information. Each Core Manual addresses a particular process or function and how it is implemented within the TYPO3 source code. These may include information on available APIs, specific configuration options, etc.

Core Manuals are written as reference manuals. The reader should rely on the Table of Contents to identify what particular section will best address the task at hand.

Table of Contents

Introduction

About this document

This document defines coding guidelines for the TYPO3 v4 project. Following these guidelines is mandatory for TYPO3 core developers and contributors to the TYPO3 core.

Extension authors are strongly encouraged to follow these guidelines when developing extensions for TYPO3. Following these guidelines makes it easier to read the code, analyze it for learning or perform code reviews. These guidelines also help to prevent typical errors in the TYPO3 code.

This document defines how TYPO3 code, files and directories should be structured and formatted. It dœs not teach how to program for TYPO3 and dœs not provide technical information about TYPO3.

Page numbers in this document refer to its printable version (available from the typo3.org web site).

What's new

The latest version of the CGLs mostly contains more complete and precise information about already existing guidelines. It also reflects changes in the coding of TYPO3 4.5, like modified XCLASS declarations and the use of UTF-8 encoding in the source code. Also mentioned are more recent changes, like the switch to Git.

Furthermore the appendix which described how to properly set up various IDEs for working with the TYPO3 v4 Core has been move to the wiki: http://wiki.typo3.org/PHP_Editors_/_IDE_for_TYPO3

Credits

The original TYPO3 coding guidelines document was written by Kasper Skårhøj. The current version is based on a complete rewrite prepared by Ingo Renner and Dmitry Dulepov in 2008. It is currently maintained by François Suter. All changes go through an approval process by the TYPO3 Core Team.

Feedback

For general questions about the documentation get in touch by writing to documentation@typo3.org.

If you find a bug in this manual, please file an issue in this manual's bug tracker: http://forge.typo3.org/projects/typo3v4-doc_core_cgl/issues

Maintaining quality documentation is hard work and the Documentation Team is always looking for volunteers. If you feel like helping please join the documentation mailing list (typo3.projects.documentation on lists.typo3.org).

Conventions used in this document

`Monospace` font is used for:

- File names and directories. Directories have slash (/) in the end of the directory name.
- Code examples
- TYPO3 module names
- Extension keys
- TYPO3 namespaces (see page 6)

TYPO3 Frontend and Backend are spelled with the first letter in uppercase because they are seen as subsystems.

File system conventions

There are certain conventions about naming files and directories in the TYPO3 core and extensions. Some of them are historical and do not follow other formal rules. They will be described separately. New core classes and extensions are required to follow formal rules outlined below.

TYPO3 directory structure

By default a TYPO3 installation consists from the following directories:

Directory	Description
fileadmin/	This is a directory where users can store files. Typically images or HTML files appear in this directory and/or its subdirectories. Often this directory is used for downloadable files. This directory is the only one accessible using the TYPO3 `File` module.
t3lib/	TYPO3 library directory
typo3/	TYPO3 Backend directory. This directory contains the Backend files. Additionally, it contains some extensions in the `ext/` (not used by the TYPO3 core) and `sysext/` directories. For example, the "`cms`" extension contains the code for generating the Frontend website.
typo3conf/	TYPO3 configuration directory
typo3conf/ext/	Directory for TYPO3 extensions
typo3temp/	Directory for temporary files. It contains subdirectories for temporary files of extensions and TYPO3 components.
uploads/	Default upload directory. For example, all images uploaded with “Image with text” content element will be in this directory. Extensions can use `uploadfolder` setting in the `ext_emconf.php` to specify extension's own upload directory inside this directory.

This structure is default for TYPO3 installation. Other non-TYPO3 applications can add their own directories.

TYPO3 files and user files

All files in the TYPO3 web site directory hierarchy are divided to TYPO3 files and user files. TYPO3 files are files that come with the TYPO3 source package released by the TYPO3 core team. This includes files inside `t3lib/` and `typo3/` directories and the file named `index.php` in the root of TYPO3 installation

All other files are user files. This includes extensions, files in the `fileadmin/` directory or files generated by TYPO3 (like thumbnails or temporary CSS files).

Extension directory structure

An extension directory contains the following files and directories:

Name	Description
ext_emconf.php	This is the only mandatory file in the extension. It describes extension to the rest of TYPO3.
ext_icon.gif	This is icon of the extension. The name may not be changed.
ext_localconf.php	This file contains hook definitions and plugin configuration. The name may not be changed.
ext_tables.php	This file contains table declarations. For more information about table declarations and definitions see the “TYPO3 Core API” document. The name may not be changed.

Name	Description

ext_tables.sql	This files contains definitions for extension tables. The name may not be changed. The file may contain either a full table definition or a partial table. The full table definition declares extension's tables. It looks like a normal SQL `CREATE TABLE` statement.
	The partial table definition contains a list of the fields that will be added to the existing table. Here is an example:
	`CREATE TABLE pages (`
	`tx_myext_field int(11) DEFAULT '0' NOT NULL,`
	`);`
	Notice the comma after the field. In the full table definition it will be a error but in the partial table definition it is required. TYPO3 will merge this table definition to the existing table definition when comparing expected and actual table definitions. Partial definitions can also contain indexes and other directives. They can also change existing table fields though it is not recommended because it may create problems with the TYPO3 core and/or other extensions.
	TYPO3 parses `ext_tables.sql` files. TYPO3 expects that all table definitions in this file look like the ones produced by the `mysqldump` utility. Incorrect definitions may not be recognized by the TYPO3 SQL parser.

tca.php	This file contains full table definitions for extension tables.
locallang*.xml	These files contains localizable labels. They can also appear in subdirectories.
doc/	This directory contains the extension manual. The name may not be changed.
doc/manual.sxw	This file contains extension manual in OpenOffice 1.0 format. The name or file format may not be changed. See the “Documentation template” document on `typo3.org` for more information about extension manuals.
piX/	These directories contain Frontend plugins. If extension is generated by the Kickstarter, *X* will be a number. It is recommended to give more meaningful names to Frontend plugin directories.
svX/	These directories contain TYPO3 services. If extension is generated by the Kickstarter, *X* will be a number. It is recommended to give more meaningful names to service directories.
modX/	These directories commonly contain Backend modules. If extension is generated by the Kickstarter, *X* will be substituted with a number. It is recommended to give more meaningful names to Backend module directories.
modfuncX/	These directories commonly contain Backend submodules (embedded into other modules). If extension is generated by the Kickstarter, *X* will be a number. It is recommended to give more meaningful names to these directories.
lib/	Directory for non-TYPO3 files supplied with extension. TYPO3 is licensed under GPL version or 2 or any later version. Any non-TYPO3 code must be compatible with GPL version 2 or any later version. Note: this name is not mandatory but recommended.

This directory structure is **strongly recommended**. Extensions may create their own directories (for example, move all language files into other directories).

File names

TYPO3 requires all PHP class file names to start with `class.` prefix followed by a namespace prefix, underscore character, class name, underscore character and extension. For information on namespaces and namespace prefix, see the next section of this document. Extension for PHP files is always `php`.

Non-class files must not start with `class.` prefix. It is recommended to use only PHP classes and avoid non-class files.

Classes that contain PHP interfaces must have `interface.` prefix.

One file can contain only one class or interface.

File names must be all lower case.

Unit test files

Unit test files are located in the "tests" folder at the root of the TYPO3 source, within a sub-structure matching the source code's structure.

The naming conventions for the files are different than those explained above:

1. the names are not prepended with "class."
2. "Test" is appended at the end of the name, before ".php"

Example

The unit test class file for `t3lib/db/class.t3lib_db_preparedstatement.php` is:

```
tests/t3lib/db/t3lib_db_preparedstatementTest.php
```

See more about unit testing in the "Unit tests" chapter.

Namespaces

TYPO3 logically separates all files and directories into several namespaces. These namespaces serve two purposes:

1. They show where a file or directory belongs inside TYPO3 CMS
2. They restrict PHP execution only to files from a certain namespace (for example, TYPO3 expects all callable functions to be in the `user_` or `tx_` namespace).

Sections below describe currently defined namespaces inside TYPO3.

t3lib

The `t3lib` namespace is reserved for common TYPO3 files. These files are used by both Frontend and Backend. Physically this namespace corresponds to the `t3lib/` directory in the TYPO3 directory hierarchy.

All PHP class files in `t3lib` name space start with `class.t3lib_` prefix.

The `t3lib/` directory contains subdirectories. Class and interface files in these subdirectories have directory name appended to the `t3lib_` prefix and separated from the class name by the underscore character. For example, files in the `t3lib/cache/` are named like `class.t3lib_cache_exception.php`. Files inside `t3lib/cache/backend` are named like `class.t3lib_cache_backend_abstractbackend.php`.

User files are not allowed inside this namespace.

typo3

This namespace is reserved for TYPO3 Backend files. No user files are allowed here.

Historically files in this namespace have different prefixes and do not follow common naming rules.

tslib

`tslib` historically stands for “TypoScript library”. This namespace is part of `cms` extension. Physically it is located in `typo3/sysext/cms/tslib/` directory and contains Frontend page and content generation files.

Files in this namespace historically may not follow common naming conventions.

User files are not allowed in this namespace.

tx_

This namespace is reserved for extensions. Extension PHP class files must start with `class.tx_` prefix, followed by the extension key without underscores, another underscore and the name of the class in lower case. The file name ends with `php` extension. For example, if extension key is `test_ext`, the file name will be `class.tx_testext_myclass.php` and the name of the class will be `tx_testext_myClass`.

User files from this namespace commonly found in `typo3conf/ext/` directory. Optionally these files can be installed to the `typo3/ext/` directory to be shared by many TYPO3 installations.

user_

This namespace is used for PHP files without PHP classes or for extensions local to a single installation. It is not recommended to create extensions with `user_` prefix.

Non-class files contain PHP functions. These functions can be called by TYPO3 only if they have `user_` prefix (the prefix can be changed by administrator in Install tool). All functions inside such files must have `user_` prefix as well. Usually such files are placed inside `fileadmin/` directory or its subdirectory.

ux_

This name space is reserved for `XCLASS` files. These files usually appear in extensions.

PHP file formatting

General requirements to PHP files

PHP tags

Each PHP file in TYPO3 must use full PHP tags. There must be exactly one pair of opening and closing tags (no closing and opening tags in the middle of the file). Example:

```
<?php
      // File content goes here
?>
```

There must be no empty lines after the closing PHP tag. Empty lines after closing tags break output compression in PHP and/or result in AJAX errors.

Line breaks

TYPO3 uses Unix line endings (`\n`, PHP `chr(10)`). If a developer uses Windows or Mac OS X platform, the editor must be configured to use Unix line endings.

Line length

Very long lines of code should be avoided for questions of readability. A line length of about 130 characters (**including** tabs) is fine. Longer lines should be split into several lines whenever possible. Each line fragment starting from the second must be indented with one more tab characters. Example:

```
$rows = $GLOBALS['TYPO3_DB']->exec_SELECTgetRows('uid, title', 'pages',
      'pid=' . $this->fullQuoteStr($this->pid, 'pages') . $this->cObj-
>enableFields('pages'),
      '', 'title');
```

or even better for readability:

```
$rows = $GLOBALS['TYPO3_DB']->exec_SELECTgetRows('uid, title',
      'pages',
      'pid=' . $this->fullQuoteStr($this->pid, 'pages') . $this->cObj-
>enableFields('pages'),
      '',
      'title'
);
```

Comment lines should be kept within a limit of about 80 characters (**excluding** tabs) as it makes them easier to read.

Notes

- tabs are considered to be 4 spaces wide.
- when splitting a line, try to split it at a point that makes as much sense as possible. In the above example, the line is split between two arguments and not in the middle of one. In case of long logical expressions, put the logical operator at the beginning of the next line, e.g.:

```
if ($GLOBALS['TYPO3_CONF_VARS']['SYS']['curlUse'] == '1'
      && preg_match('/^(?:http|ftp)s?|s(?:ftp|cp):/', $url)) {
```

Whitespace and indentation

TYPO3 uses tab characters to indent source code. One indentation level is one tab.

There must be no white spaces in the end of a line. This can be done manually or using a text editor that takes care of this (see section “Settings for editors” on page Error: Reference source not found).

Spaces must be added:

- on both sides of string, arithmetic, assignment and other similar operators (for example, `.`, `=`, `+`, `-`, `?`, `:`, `*`, etc)
- after commas
- in single line comments after the comment sign (double slash)

– after asterisks in multiline comments

Character set

All TYPO3 source files use the UTF-8 character set since version 4.5. Files from third-party libraries may have different encodings.

File structure

TYPO3 files use the following structure:

1. Opening PHP tag
2. Copyright notice
3. File information block (with optional function index) in phpDoc format
4. Included files
5. Class information block in phpDoc format
6. PHP class
7. XCLASS declaration
8. Optional module execution code (for example, in eID classes)
9. Closing PHP tag

The following sections discuss each of these parts.

Copyright notice

TYPO3 is released under the terms of GNU General Public License version 2 or any later version. The copyright notice with a reference to the GPL must be included at the top of every TYPO3 PHP class file. `user_` files must have this copyright notice as well. Example:

```
<?php
/***************************************************************
*  Copyright notice
*
*  (c) YYYY Your name here (your@email.here)
*  All rights reserved
*
*  This script is part of the TYPO3 project. The TYPO3 project is
*  free software; you can redistribute it and/or modify
*  it under the terms of the GNU General Public License as published by
*  the Free Software Foundation; either version 2 of the License, or
*  (at your option) any later version.
*
*  The GNU General Public License can be found at
*  http://www.gnu.org/copyleft/gpl.html.
*  A copy is found in the textfile GPL.txt and important notices to the license
*  from the author is found in LICENSE.txt distributed with these scripts.
*
*
*  This script is distributed in the hope that it will be useful,
*  but WITHOUT ANY WARRANTY; without even the implied warranty of
*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*  GNU General Public License for more details.
*
*  This copyright notice MUST APPEAR in all copies of the script!
***************************************************************/
```

This notice may not be altered except for the year, author name and author e-mail.

File information block

File information block follows the copyright statement and provides basic information about the file. It should include file name, description of the file and information about the author (or authors). Example:

```
/**
 * class.tx_myext_pi1.php
 *
```

```
 * Provides XYZ plugin implementation.
 *
 * @author John Doe <john.doe@example.com>
 */
```

The file information block can also contain the optional function index. This index is created and updated by the `extdeveval` extension.

Included files

Files are included using `require_once` function. All TYPO3 files must use absolute paths in calls to `require_once`. There are two ways to obtain the path to the included file:

1. Use one of the predefined TYPO3 constants: `PATH_tslib, PATH_t3lib, PATH_typo3, PATH_site`. The first three contain absolute paths to the corresponding TYPO3 directories. The last constant contains absolute path to the TYPO3 root directory. Example:

   ```
   require_once(PATH_tslib . 'class.tslib_pibase.php');
   ```

2. Use `t3lib_extMgm::extPath()` function. This function accepts two arguments: extension key and path to the included file. The second argument is optional but recommended to use. Examples:

   ```
   require_once(t3lib_extMgm::extPath('lang', 'lang.php'));
   require_once(t3lib_extMgm::extPath('lang') . 'lang.php');
   ```

Always use one of these two ways to include files. This is required to include files even from the current directory. Some installations do not have the current directory in the PHP include path and `require_once` without a proper path will result in fatal PHP error.

Class information block

Class information block is similar to the file information block and describes the class in the file. Example:

```
/**
 * This class provides XYZ plugin implementation.
 *
 * @author John Doe <john.doe@example.com>
 * @author Jane Doe <jane.doe@example.com>
 */
```

PHP class

PHP class follows the Class information block. PHP code must be formatted as described in chapter “PHP syntax formatting” on page 11.

The class name is expected to follow some conventions. The namespace and path parts (see “Namespaces” on page 6) are all lowercase and separated by underscores (“_”). At the end comes the “true” class name which must be written in upper camel case.

Taking again the example of file `class.t3lib_cache_backend_abstractbackend.php`, the PHP class declaration will look like:

```
class t3lib_cache_backend_AbstractBackend {
      …
}
```

XCLASS declaration

The XCLASS declaration must follow the PHP class. The format of the XCLASS is very important. Please follow the example below, otherwise the TYPO3 Extension Manager will complain about a missing XCLASS declaration.

The XCLASS declaration must include proper path to the current class file. The following example assumes that extension key is `myext`, file name is `class.tx_myext_pi1.php` and file is located in the `pi1` subdirectory of the extension:

```
if (defined('TYPO3_MODE') && isset($GLOBALS['TYPO3_CONF_VARS'][TYPO3_MODE]['XCLASS']
['ext/myext/pi1/class.tx_myext_pi1.php'])) {
      include_once($GLOBALS['TYPO3_CONF_VARS'][TYPO3_MODE]['XCLASS']
['ext/myext/pi1/class.tx_myext_pi1.php']);
}
```

Optional module execution code

Module execution code instantiates the class and runs its method(s). Typically this code can be found in eID scripts and old Backend modules. Here is how it may look like:

```
$controller = t3lib_div::makeInstance('tx_myext_ajaxcontroller');
$controller->main();
```

This code must appear **after** the XCLASS declaration. `$SOBE` is traditional but not required name.

PHP syntax formatting

Identifiers

All identifiers must use `camelCase` and start with a lower case letter. Underscore characters are not allowed. Abbreviations should be avoided. Examples of good identifiers:

```
$goodName
$anotherGoodName
```

Examples of bad identifiers:

```
$BAD_name
$unreasonablyLongNamesAreBadToo
$noAbbrAlwd
```

The lower camel-case rule also applies to acronyms. Thus:

```
$someNiceHtmlCode
```

is correct, whereas

```
$someNiceHTMLCode
```

is not.

In particular the abbreviations "FE" and "BE" should be avoided and the full "Frontend" and "Backend" words used instead.

Identifier names must be descriptive. However it is allowed to use traditional integer variables like `$i`, `$j`, `$k` in `for` loops. If such variables are used, their meaning must be absolutely clear from the context where they are used.

The same rules apply to functions and class methods. Examples:

```
protected function getFeedbackForm()
public function processSubmission()
```

Class constants should be clear about what they define. Correct:

```
const USERLEVEL_MEMBER = 1;
```

Incorrect:

```
const UL_MEMBER = 1;
```

Variables on the global scope may use upper case and underscore characters.

Examples:

```
$TYPO3_CONF_VARS
$TYPO3_DB
```

Comments

Comments in the code are highly welcome and recommended. Inline comments must precede the commented line and be indented by one tab. Example:

```
protected function processSubmission() {
            // Check if user is logged in
      if ($GLOBALS['TSFE']->fe_user->user['uid']) {
            …
      }
}
```

Comments must start with "//". Starting with "#" is not allowed.

Class constants and variable comments should follow PHP doc style and precede the variable. Variable type must be specified for non-trivial types and optional for trivial types. Example:

```
/** Number of images submitted by user */
protected $numberOfImages;
/**
 * Local instance of tslib_cObj class
 *
 * @var tslib_cObj
 */
protected $localCobj;
```

Single line comments are allowed when there is no type declaration for the class variable or constant.

If a variable can hold values of different types, use `mixed` as type.

Debug output

During development it is allowed to use `debug()` or `t3lib_div::debug()` function calls to produce debug output. However all debug statements must be removed (removed, not commented!) before committing the code to the Subversion repository.

Curly braces

Usage of opening and closing curly braces is mandatory in all cases where they can be used according to PHP syntax (except `case` statements).

The opening curly brace is always on the same line as the preceding construction. There must be a space (not a tab!) before the opening brace. The opening brace is always followed by a new line.

The closing curly brace must start on a new line and be indented to the same level as the construct with the opening brace. Example:

```
protected function getForm() {
	if ($this->extendedForm) {
		// generate extended form here
	} else {
		// generate simple form here
	}
}
```

The following is not allowed:

```
protected function getForm()
{
	if ($this->extendedForm) { // generate extended form here
	} else {
		// generate simple form here
	}
}
```

Conditions

Conditions consist from `if`, `elseif` and `else` keywords. TYPO3 code must not use the `else if` construct.

The following is the correct layout for conditions:

```
if ($this->processSubmission) {
	// Process submission here
} elseif ($this->internalError) {
	// Handle internal error
} else {
	// Something else here
}
```

Here is an example of the incorrect layout:

```
if ($this->processSubmission) {
	// Process submission here
}
```

```
elseif ($this->internalError) {
	// Handle internal error
} else // Something else here
```

It is recommended to create conditions so that shortest block gœs first. For example:

```
if (!$this->processSubmission) {
	// Generate error message, 2 lines
} else {
	// Process submission, 30 lines
}
```

If the condition is long, it must be split into several lines. Each condition on the line starting from the second should be indented with a two or more indents relative to the first line of the condition:

```
if ($this->getSomeCodition($this->getSomeVariable()) &&
		$this->getAnotherCondition()) {
	// Code follows here
}
```

Ternary conditional operator must be used only if it has two outcomes. Example:

```
$result = ($useComma ? ',' : '.');
```

Wrong usage of ternary conditional operator:

```
$result = ($useComma ? ',' : $useDot ? '.' : ';');
```

Assignment in conditions should be avoided. However if it makes sense to do assignment in condition, it should be surrounded by the extra pair of brackets. Example:

```
if (($fields = $GLOBALS['TYPO3_DB']->sql_fetch_assoc($res))) {
	// Do something
}
```

The following is allowed but not recommended:

```
if (FALSE !== ($fields = $GLOBALS['TYPO3_DB']->sql_fetch_assoc($res))) {
	// Do something
}
```

The following is not allowed (missing the extra pair of brackets):

```
while ($fields = $GLOBALS['TYPO3_DB']->sql_fetch_assoc($res)) {
	// Do something
}
```

Switch

`case` statements are indented with a single indent (tab) inside the `switch` statement. The code inside the `case` statements is further indented with a single indent. The `break` statement is aligned with the code. Only one `break` statement is allowed per `case`.

The `default` statement must be the last in the `switch` and must not have a `break` statement.

If one `case` block has to pass control into another `case` block without having a `break`, there must be a comment about it in the code.

Examples:

```
switch ($useType) {
	case 'extended':
		$content .= $this->extendedUse();
		// Fall through
	case 'basic':
		$content .= $this->basicUse();
		break;
	default:
		$content .= $this->errorUse();
}
```

Loops

The following loops can be used:

- do
- while
- for
- foreach

The use of each is not allowed in loops.

for loops must contain only variables inside (no function calls). The following is correct:

```
$size = count($dataArray);
for ($element = 0; $element < $size; $element++) {
      // Process element here
}
```

The following is not allowed:

```
for ($element = 0; $element < count($dataArray); $element++) {
      // Process element here
}
```

do and while loops must use extra brackets if assignment happens in the loop:

```
while (($fields = $GLOBALS['TYPO3_DB']->sql_fetch_assoc($res))) {
      // Do something
}
```

There's a special case for foreach loops when the value is not used inside the loop. In this case the dummy variable $_ (underscore) is used:

```
foreach ($GLOBALS['TCA'] as $table => $_) {
      // Do something with $table
}
```

This is done for performance reasons, as it is faster than calling array_keys() and looping on its result.

Strings

All strings must use single quotes. Double quotes are allowed only to create the new line character ("\n").

String concatenation operator must be surrounded by spaces. Example:

```
$content = 'Hello ' . 'world!';
```

However the space after the concatenation operator must not be present if the operator is the last construction on the line. See the section about white spaces on page 8 for more information.

Variables must not be embedded into strings. Correct:

```
$content = 'Hello ' . $userName;
```

Incorrect:

```
$content = "Hello $userName";
```

Multiline string concatenations are allowed. Line concatenation operator must be at the end of the line. Lines starting from the second must be indented relative to the first line. It is recommended to indent lines one level from the start of the string on the first level:

```
$content = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. ' .
                'Donec varius libero non nisi. Proin eros.';
```

Booleans

Booleans must use PHP's language constructs and not explicit integer values like 0 or 1. Furthermore they should be written in uppercase, i.e. TRUE and FALSE.

Arrays

Array declarations use the "array" keyword in lower case, with no blank between it and the opening bracket. Thus:

```
$a = array();
```

Array components are declared each on a separate line. Such lines are indented with one more tab than the start of the declaration. The closing bracket is on the same indentation level as the variable. Every line containing an array item ends with a comma. This may be omitted if there are no further elements, at the developer's choice. Example:

```
$thisIsAnArray = array(
	'foo' => 'bar',
	'baz' => array(
		0 => 1
	)
);
```

Nested arrays follow the same pattern. This formatting applies even to very small and simple array declarations, e.g.

```
$a = array(
	0 => 'b',
);
```

NULL

Similarly this special value is written in uppercase, i.e. `NULL`.

PHP5 features

The use of PHP5 features is strongly recommended for extensions and mandatory for the TYPO3 core versions 4.2 or greater.

Class functions must have access type specifier: `public`, `protected` or `private`. Notice that `private` may prevent XCLASSing of the class. Therefore `private` can be used only if it is absolutely necessary.

Class variables must use access specifier instead of `var` keyword.

Type hinting must be used when function expects array or an instance of a certain class. Example:

```
	protected function executeAction(tx_myext_action& $action, array $extraParameters) {
		// Do something
	}
```

Static functions must use `static` keyword. This keyword must be the first keyword in the function definition:

```
	static public function executeAction(tx_myext_action& $action, array $extraParameters)
{
		// Do something
	}
```

`abstract` keyword also must be on the first position in the function declaration:

```
	abstract protected function render();
```

Global variables

Use of `global` is not recommended. Always use `$GLOBALS['variable']`.

Functions

If a function returns a value, it must always return it. The following is not allowed:

```
function extendedUse($enabled) {
	if ($enabled) {
		return 'Extended use';
	}
}
```

The following is the correct behavior:

```
function extendedUse($enabled) {
	$content = '';
	if ($enabled) {
		$content = 'Extended use';
	}
```

```
        return $content;
}
```

In general there should be a single return statement in the function (see the preceding example). However a function can return during parameter validation before it starts its main logic. Example:

```
function extendedUse($enabled, tx_myext_useparameters $useParameters) {
        // Validation
        if (count($useParameters->urlParts) < 5) {
                return 'Parameter validation failed';
        }

        // Main functionality
        $content = '';
        if ($enabled) {
                $content = 'Extended use';
        } else {
                $content = 'Only basic use is available to you!';
        }
        return $content;
}
```

Functions should not be long. "Long" is not defined in terms of lines. General rule is that function should fit into $^2/_3$ of the screen. This rule allows small changes in the function without splitting the function further. Consider refactoring long functions into more classes or methods.

Using phpDoc

phpDoc is used for documenting source code. Typically TYPO3 code uses the following phpDoc keywords:

- `@author`
- `@access`
- `@global`
- `@param`
- `@package`
- `@return`
- `@see`
- `@subpackage`
- `@var`
- `@deprecated`

For more information on phpDoc see the phpDoc web site at `http://www.phpdoc.org/`

TYPO3 requires that each class, function and method be documented with phpDoc. For information on phpDoc use for classes declarations see "Class information block" on page 10.

Note that the `@author` tag should **not** be used in function or method phpDoc comment blocks – only at class level – because it is too liable to change frequently and authors would accumulate indefinitely. `git blame` is enough for tracking changes.

Function information block

Functions should have parameters and return type documented. Example:

```
/**
 * Initializes the plugin.
 *
 * Checks the configuration and substitutes defaults for missing values.
 *
 * @param array $conf Plugin configuration from TypoScript
 * @return boolean TRUE if initialization was successful, FALSE otherwise
 * @see tx_myext_class:anotherFunc()
```

```
 */
protected function initialize(array $conf) {
      // Do something
}
```

Short and long description

A method or class may have both a short and a long description. The short description is the first piece of text inside the phpDoc block. It ends with the next blank line. Any additional text after that line and before the first tag is the long description.

Use `@return void` when a function dœs not return a value.

The ChangeLog file

The TYPO3 core comes with a `ChangeLog` file where all changes are recorded. This used to be maintained manually, but will be generated automatically since the move to the Git version-control system of March 1, 2011.

Coding: best practices

This section documents best practices when developing for TYPO3.

Accessing the database

The TYPO3 database should be always accessed through the use of `$GLOBALS['TYPO3_DB']`. This is the instance of `t3lib_db` class from `t3lib/class.t3lib_db.php`.

The same rule applies for accessing non-TYPO3 databases: they should be accessed by using a different instance of the same class. Failing this condition may corrupt TYPO3 database or prevent access to TYPO3 database for the rest of the script.

Singletons

TYPO3 supports singleton pattern for classes. Singletons are instantiated only once per HTTP request regardless of the number of calls to the `t3lib_div::makeInstance()`. To use singleton pattern class must implement `t3lib_Singleton` interface:

```
require_once(PATH_t3lib . 'interfaces/interface.t3lib_singleton.php');

class tx_myext_mySingletonClass implements t3lib_Singleton {
      …
}
```

This interface has no methods to implement.

Static methods

When a given class calls one of its own static methods (or from one of its parents), the code should use the `self` keyword instead of the class name.

Example

```
class tx_myext_MyClass {
    public static function methodA() {
      //...
    }
    public static function methodB() {
      self::methodA(); // instead of tx_myext_MyClass::methodA();
    }
}
```

Localization

TYPO3 is designed to be fully localizable. Hard-coded strings should thus be avoided unless there are some technical limitations (e.g. some early or low-level stuff where a `lang` object is not available).

Defining the localized strings

Here are some rules to respect when working with labels in locallang files:

- always check the existing locallang files to see if a given localized string already exists, in particular `EXT:lang/locallang_common.xml` and `EXT:lang/locallang_core.xml`.
- localized strings should never be all uppercase. If this is needed, then appropriate methods to make them uppercase should be used where needed.
- localized strings must not be split into several parts to include stuff in their middle. Rather use a single string with `sprintf()` markers (`%s`, `%d`, etc.).
- when a localized string contains several `sprintf()` markers, it **must** use numbered arguments (e.g. `%1$d`).
- punctuation marks **must** be included in the localized string – including trailing marks – as different punctuation marks (e.g. "?" and "¿") may be used in various languages. Also some

languages include blanks before some punctuation marks.

- localized strings are not supposed to contain HTML tags, except for CSH. They should be avoided whenever possible.

Once a localized string appears in a released version of TYPO3, it cannot be changed (unless it needs grammar or spelling fixes). Nor can it be removed. If the label a localized string has to be changed, a new one should be introduced instead.

Using the localized strings

Localized string are displayed using the available API, mostly `lang::getLL()` when the corresponding locallang file is loaded and `lang::sL()` otherwise. In both these methods, the second call parameter should be left out, unless there's a compelling reason to set it to `TRUE` (which triggers the use of `htmlspecialchars()`).

Unit tests

Using unit tests

Although the coverage is far from complete, there are already quite a lot of unit tests for the TYPO3 Core. Anytime something is changed in the Core, all existing unit tests must be run to ensure that nothing was broken.

Adding unit tests

The use of unit tests is strongly encouraged. Every time a new feature is introduced or an existing one is modified, a unit test should be added.

Handling deprecation

This section describes the rules to follow for removing existing functions or parameters from TYPO3. The general principle is that functions or parameters are removed **two major versions** after they were set to be deprecated.

To start the deprecation process for a parameter of a TYPO3 core function, please mark it within the phpDoc param part:

```
/**
 * ...
 * @param   string      DEPRECATED since TYPO3 4.X - is not used anymore because...
 * ...
 */
```

For a whole function inside one of the TYPO3 core classes, use the phpDoc `@deprecated` parameter:

```
/**
 * ...
 * @return...
 * @deprecated since TYPO3 4.X - is not used anymore, use FUNCNAME instead
 */
```

Anyone can submit a patch to remove deprecated elements starting with version TYPO3 4.X+2.