



# **DEVELOPER CODING STYLE GUIDELINES**

**1<sup>st</sup> Draft prepared by Bermi Ferrer**

**August 2, 2006**

## Using this Standard.

If you want to make a local copy of this standard and use it as your own you are perfectly free to do so, but keep in mind that many rules apply only to Akelos Framework environments

# Introduction

This document defines how the PHP code in the Akelos Framework is formatted.

The Akelos Coding Standards apply to code that is part of the official Akelos Framework distribution.

Naming conventions are also defined in this document. The code in the Akelos Framework must follow this standard to ensure a coherent looking code base.

This document has evolved from <http://www.possibility.com/Cpp/CppCodingStandard.html> and <http://pear.php.net/manual/en/standards.php>

## Standardization is Important

It helps if the standard annoys everyone in some way so everyone feels they are on the same playing field. The proposal here has evolved over many projects, many companies, and literally a total of many weeks spent arguing. It is no particular person's style and is certainly open to local amendments.

### Good Points

When a project tries to adhere to common standards a few good things happen: programmers can go into any code and figure out what's going on

- new people can get up to speed quickly
- people new to PHP are spared the need to develop a personal style and defend it to the death
- people new to PHP are spared making the same mistakes over and over again
- people make fewer mistakes in consistent environments
- programmers have a common enemy :-)

### Bad Points

Now the bad:

- the standard is usually stupid because it's not what I do
- standards reduce creativity
- standards are unnecessary as long as people are consistent
- standards enforce too much structure
- people ignore standards anyway

## **Discussion**

The experience of many projects leads to the conclusion that using coding standards makes the project go smoother. Are standards necessary for success? Of course not. But they help, and we need all the help we can get! Be honest, most arguments against a particular standard come from the ego. Few decisions in a reasonable standard really can be said to be technically deficient, just matters of taste. So be flexible, control the ego a bit, and remember any project is fundamentally a team effort.

## **Interpretation**

### **Conventions**

The use of the word "shall" in this document requires that any project using this document must comply with the stated standard.

The use of the word "should" directs projects in tailoring a project-specific standard, in that the project must include, exclude, or tailor the requirement, as appropriate.

The use of the word "may" is similar to "should", in that it designates optional requirements.

### **Standards Enforcement**

First, any serious concerns about the standard should be brought up and worked out within the group. Maybe the standard is not quite appropriate for your situation. It may have overlooked important issues or maybe someone in power vehemently disagrees with certain issues :-).

In any case, once finalized hopefully people will play the adult and understand that this standard is reasonable, and has been found reasonable by many other programmers, and therefore is worthy of being followed even with personal reservations.

Failing willing cooperation it can be made a requirement that this standard must be followed to pass a code inspection.

Failing that the only solution is a massive tickling party on the offending party.

## **Accepting an Idea**

11. It's impossible.
12. Maybe it's possible, but it's weak and uninteresting.
13. It is true and I told you so.
14. I thought of it first.
15. How could it be otherwise?

If you come to objects with a negative preconception please keep an open mind. You may still conclude objects are bunk, but there's a road you must follow to accept something different. Allow yourself to travel it for a while.

# Header Comment Blocks

All source code files in the Akelos distribution must contain the following comment blocks as the header:

Example for LGPL'ed Akelos code:

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

// +-----
// | Akelos Framework - http://www.akelos.org
// |
// +-----
// | Copyright (c) 2002-2006, Akelos Media, S.L. & Bermi Ferrer Martinez
// |
// | Released under the GNU Lesser General Public License, see
// | LICENSE.txt|
// +-----
//
+

/**
 * @package AkelosFramework
 * @subpackage NameOfThePackage
 * @author Your name <myname@example.com>
 * @copyright Copyright (c) 2002-2006, Akelos Media, S.L.
http://www.akelos.org
 * @license GNU Lesser General Public License
<http://www.gnu.org/copyleft/lesser.html>
 */
```

# Naming Conventions

## Make Names Fit

Names are the heart of programming. In the past people believed knowing someone's true name gave them magical power over that person. If you can think up the true name for something, you give yourself and the people coming after power over the code. Don't laugh!

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

If you find all your names could be Thing and DoIt then you should probably revisit your design.

## Class Names

- Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.
- Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. Try a CRC card session to see if your objects have more responsibilities than they should.
- Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.
- Suffixes are sometimes helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

## Method and Function Names

- Usually every method and function performs an action, so the name should make clear what it does: checkForErrors() instead of errorCheck(), dumpDataToFile() instead of dataFile(). This will also make functions and data objects more distinguishable.
- Suffixes are sometimes useful:
  - *max* - to mean the maximum value something can have.
  - *key* - key value.

For example: retryMax to mean the maximum number of retries.

Prefixes are sometimes useful:

- *is* and *has* - to ask a question about something. Whenever someone sees *is* they will know it's a question.
- *get* - get a value.
- *set* - set a value.

For example: isHitRetryLimit.

## No All Upper Case Abbreviations

- When confronted with a situation where you could use an all upper case abbreviation instead use an initial upper case letter followed by all lower case letters. No matter what. Do use: getHtmlStatistic.  
Do not use: getHTMLStatistic.

### Justification

- People seem to have very different intuitions when making names containing abbreviations. It's best to settle on one strategy so the names are absolutely predictable. Take for example NetworkABCKey. Notice how the C from ABC and K from key are confused. Some people don't mind this and others just hate it so you'll find different policies in different code so you never know what to call something.

### Example

```
fluidOz           // NOT fluidOZ  
getHtmlStatistic // NOT getHTMLStatistic
```

# Class Library Names

Libraries (*any file located in the /lib/ directory of the application*) should be named with capital letters at the beginning of each word. Use '**CamelCase**' for naming; a session cache class would be stored in */lib/SessionCache.php*.

If the library/class is extended, the extending files should be stored in a directory under */lib/* with the same name as the original library. Subclasses follow the exact same naming requirements, except that if the subclass is instantiated by a factory method, it should be all lowercase.

As there are no name spaces in PHP 4.x, Akelos Framework Core classes must be prefixed using the unique string *Ak*. This will prevent class name conflicts among libraries from different vendors and groups.

**Akelos complete data structure library could use Ak as a prefix**, so classes would be:

```
class AkContactManager
{
}
```

# Class Names

- Use upper case letters as word separators, lower case for the rest of a word
- First character in a name is upper case
- No underbars ('\_')
- Akelos Framework Core classes must be prefixed using the unique string Ak. This will prevent class name conflicts among libraries from different vendors and groups.

## Justification

- Of all the different naming strategies many people found this one the best compromise.

```
class ContactManager
{
}
```

```
class AkXmlParser
{
}
```

# Class Attributes Names

Class member attribute names use the same rules as for class names but with first character in lowercase.

This allows to quickly identify attributes from methods.

The only exception of this rule is when a member attribute must hold an object. In that case the attribute name must be the same as the instanced class.

## Public attributes

You should avoid accessing attributes directly. In case you need to define a public attribute it must look like this.

```
class FileManager
{
    var $fileName = ''; // Public attribute
    var $fileOwner;    // Public attribute
    var $FileHandler; // Public attribute $this->FileHandler
                    // is an instance of FileHandler class
    ...
}
```

Some exceptions apply and underscored attributes might be used, like on ActiveRecord (column names)

## Private Attributes

```
class FileManager
{
    var $_fileName = ''; // Private attribute
    var $_fileOwner;    // Private attribute
    var $_FileHandler; // Private attribute $this->_FileHandler
                    // is an instance of FileHandler class
    ...
}
```

# Method Names

## Public Methods

Uses the same rules as for class names but with first character in lowercase.

```
class ContactManager
{
    function addContact() {};
    function activateContact() {};
}
```

Some exceptions apply and underscored methods might be used, mainly on helpers, so it can be easier to port Rails views to the Akelos Framework.

## Private Methods

Methods used exclusively within a class should begin with an underscore ('\_') character.

```
class ContactManager
{
    function addContact() {}; // This is a public method
    function _logRequest() {}; // This is a private method
    function _someHelperMethod() {}; // This is a private method
}
```

# Method Argument Names

The first character should be lower case, except if it is an Object instance.

All word beginnings after the first letter should be upper case as with class names.

If a variable is passed by reference, first character must be an underscore '\_';

This way you can always tell which variables are passed in variables and if they are a copy or a reference.

```
class XmlParser
{
    function loadXmlFile($fileName, &$_FileHandler)
    {
        // $fileName is a normal argument
        // $_FileHandler is an Object passed by reference.
        ...
    }

    function _cleanUpXml(&$_xmlFile)
    {
        // $_xmlFile is an argument passed by reference.
    }

    function getParserCopy($XmlParser)
    {
        // $XmlParser is an argument that contains an Object.
    }
}
```

# Function Definitions

- For PHP functions use the C GNU convention of all lower case letters with '\_' as the word delimiter. This makes functions very different from any class related names.
- Function declarations follow the "one true brace" convention.
- 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.
- Functions used only in the current script/class (e.g. private member methods) should begin with a \_ character (e.g. \_example\_function). This helps distinguish these private function calls from other, public function calls.
- All functions should be placed in classes unless there is a good reason not to.
- Functions should be short and do one thing. The length of a function depends on the complexity of the function, the more complex the function is the shorter should the function be. If it gets too complex then divide the function up into helper functions.
- The number of local variables in a function should not be larger than 7. 7 is a known limit to the number of things people can control at one time. Split up the function if it gets too large.
- Default parameters should be used with care.
- You should let the parameter line go over several lines with grouping of the similar parameters, if it gets long.
- Akelos Framework Core functions must be prefixed using the unique string ak\_. Or added to the Static class Ak (/lib/Ak.php) which acts as a namespace for the core functions. This will prevent function name conflicts among libraries from different vendors and groups.

```
function ak_log_error($error_code, $log_method = 'file_based')
{
    ...
}

class Ak
{
    ...
    function log_error($error_code, $log_method = 'file_based')
    {
        ...
    }
}

function _log_error_to_file($error)
{
    ...
}
```

# Variable Names

Use all lower case letters (except for variables holding objects).

Use '\_' as the word separator.

With this approach the scope of the variable is clear in the code.

Now all variables look different and are identifiable in the code.

```
class ErrorHandler
{
    function handleError($errorNumber)
    {
        $Error_handler = new OsError();
        $time_of_error = $Error_handler->getTimeOfError();
        $error_processor = $Error_handler->getErrorProcessor();
    }
}
```

# Array Elements

Array element names follow the same rules as a variable.  
Use single quotes when using non integer keys.

```
$myarr = array();  
$myarr['foo_bar'] = 'Hello'; // Good  
$myarr = array('foo_bar' => 'Hello'); // Good  
  
$myarr[foo_bar] = 'Hello'; // Bad  
  
echo $myarr['foo_bar']." world"; // Good  
echo "{$myarr['foo_bar']} world"; // Good  
  
echo $myarr[foo_bar]." world"; // Bad  
echo "$myarr[foo_bar] world"; // Bad  
echo "$myarr['foo_bar'] world"; // Bad
```

# Global Constants

Surprisingly enough, [define\(\)](#) is a somewhat slow function in PHP (as of PHP 4.3.x) so excessive use is discouraged.

Global constants should be all caps with '\_' separators. It's tradition for global constants to be named this way.

You must be careful to not conflict with other predefined global. All Akelos library constants must be prefixed with AK\_.

Constants can only contain scalar data (Boolean, integer, float and string).

Constants are mainly used in the Akelos Framework for defining whenever a class has been already declared or not.

```
if(!defined("AK_LIB_SESSION_CLASS")){
    define("AK_LIB_SESSION_CLASS", true);
    class AkSession
    {
        ... // Class declaration
    }
}
```

This technique allows you override a core Class functionality by declaring the modified Class before including the core file. This way it doesn't trigger a compilation error.

# Quotes

You should always use single quote (') characters around strings, except where double quote (") characters are required. All literal strings should be in single quotes. A comparison of single and double quote usage follows:

## Single Quotes

- Variables in the string are not parsed or expanded.
- New line symbols can be included as literal line ends (not recommended).
- To include a single quote character, escape it with a \ (backslash) character, as in:

```
echo 'Here\'s an example';
```

- To specify a \ (backslash) character, double it:

```
echo 'c:\\temp';
```

## Double Quotes

- Parses and expands variables in the string.
- Uses advanced ([sprintf](#)-style) escape sequences like \n, \\$, \t, etc.
- Use with care, as many correct looking strings are really invalid.

---

The following are all **incorrect**

```
echo "Today is the $date['day'] of $date['month']";  
$_SESSION['index'] = $_SESSION["old_index"];
```

# Control Structures

PHP has different syntax rules for using control structures, the syntax below is preferred.

- Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.
- Do not omit the curly braces under any circumstance.

---

## if/else statement

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

Multi-line if conditions are braced this way:

```
if ((condition1) || (condition2) || (condition3) ||
    (condition4)) {
    action1;
}
```

---

## ?: (if/else)

- Put the condition in parens so as to set it off from other code.
- If possible, the actions for the test should be simple functions.
- Put the action for the then and else statement on a separate line unless it can be clearly put on one line.

```
// Short way
(condition) ? funct1() : func2();

// Long way
(condition)
    ? long statement
    : another long statement;
```

---

## switch statements

- The default case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a block.

```
switch (condition) {
    case 1 : {
```

```

        action1;
    } break;

    case 2 : {
        action2;
    } break;

    default : {
        defaultaction;
    } break;
}

```

---

## for loops

- Avoid evaluating function inside loops. This slows down code.

```

// SLOW CODE
// $i<sizeof($arr) is evaluated within the loop multiple times
for ($i=0; $i<sizeof($arr); $i++) {
}
// FAST CODE
// $i<sizeof($arr) is evaluated only once
for ($i=0, $max = sizeof($arr); $i<$max; $i++) {
}

```

---

## foreach loops

- use \$k for array keys and \$v for values if a descriptive name doesn't suit

```

foreach ($cars as $car_id=>$car_details) {
}
foreach ($array as $v) {
}
foreach ($array as $k=>$v) {
}

```

- foreach makes a copy of the object that is being iterated, so technique used on the framework is to iterate over the keys

```

foreach (array_keys($objects) as $k) {
    $v =& $objects[$k];
}

```

---

## while loops

```

$i = 0;
while ( $i < 100 ){
    ...
    $i++;
}

```

# Indentation/Tabs/Space Policy

- Indent using 4 spaces for each level.
- Do not use tabs, use spaces. Most editors can substitute spaces for tabs.
- Indent as much as needed, but no more. There are no arbitrary rules as to the maximum indenting level. If the indenting level is more than 4 or 5 levels you may think about factoring out code.

## Justification

- When people using different tab settings the code is impossible to read or print, which is why spaces are preferable to tabs.
- Nobody can ever agree on the correct number of spaces, just be consistent. In general people have found 4 spaces per indentation level workable.
- As much as people would like to limit the maximum indentation levels it never seems to work in general. We'll trust that programmers will choose wisely how deep to nest code.

```
function do_something()
{
    if ($condition1){
        if ($condition2){
            while ($condition3){
            }
        }
    }
}
```

# Line Breaks

Only use UNIX style of line break (`\n`), not Windows/DOS/Mac style (`\r\n`).

Using vim, to convert from DOS style type:

```
:set ff=unix
```

Using vi, to convert from DOS style type:

```
:g/^M/s///g
```

(Note that the M is a control character, and to reproduce it when you type in the vi command you have to pad it first using the special V character.)

# PHP Code Tags

Always use `<?php ?>` to delimit PHP code, not the `<? ?>` shorthand.

This is the most portable way to include PHP code on differing operating systems and setups.

In templates, make sure to use this as well (`<?php echo $varname; ?>`), as the shortcut version (`<?=$var;?>`) does not work with [short\\_open\\_tag](#) turned off.

On views you can use the Sintags syntax <http://www.bermi.org/projects/sintags> this way you can write `{var}` and it will convert it to `<?php echo $var; ?>`

# PHP File Extensions

There are lots of different extension variants on PHP files (.html, .php, .php3, .php4, .phtml, .inc, .class...).

- Always use the extension .php.
- Always use the extension .php for your class and function libraries.

## **Justification**

- The use of .php makes it possible to enable caching on other files than .php.
- The use of .inc or .class can be a security problem. On most servers these extensions aren't set to be run by a parser. If these are accessed they will be displayed in clear text.

# 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`.

- If you are including a static filename, such as a configuration file or a template that is always used, use `require/require_once`.
- If you are dynamically including a filename, or want the code to only be used conditionally (an optional template), use `include/include_once`.

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

# Braces {} Policy

Traditional Unix policy of placing the initial brace on the same line as the keyword and the trailing brace inline on its own line with the keyword.

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

Do not omit the curly braces under any circumstance.

```
if ($condition) {  
    ...  
}  
  
while ($condition) {  
    ...  
}
```

# Parenthesis () with Key Words and Functions Policy

Keywords are not functions. By putting parenthesis next to keywords, keywords and function names are made to look alike. So following rules apply when using parenthesis.

- Do not put parenthesis next to keywords. Put a space between.
- Do put parenthesis next to function names.
- Do not use parenthesis in return statements, include, include\_once, require and require\_once when it's not necessary.

```
if (condition) {  
}  
  
while (condition) {  
}  
  
send_mail($to, $message);  
  
return true;  
  
include_once './site_config.php';
```

# Comments and inline API documentation.

## API Comments

Inline documentation for classes and functions should follow the [PHPDoc](#) convention, similar to Javadoc.

Quick example for private variable definition for Akelos:

```
/**
 * Variable description.
 *
 * @var datatype $variable name
 */
```

Quick example Class definition for Akelos:

```
/**
 * The description of the class goes here.
 *
 * @package AKELOS
 * @subpackage SubpackageName (Database, IO, Template...)
 * @author Original Author Name
 * @author Author Name <author-at-akelos.com>
 * @copyright Copyright (c) 2002-2005, Akelos Media, S.L.
 * http://www.akelos.com
 * @example PATH/TO/THE/EXAMPLE/FILE.php [if available]
 * @license GNU Lesser General Public License
 * <http://www.gnu.org/copyleft/lesser.html>
 * @since [Only if necessary - use if function is added to the
 * current release versions to indicate that the function has not been
 * available in previous versions.]
 */
```

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.

Quick example function definition for Akelos:

```
/**
 * The description of the function goes here.
 *
 * @access [public | private]
 *
 * @param datatype $variablename Description of variable.
 * @param datatype $variable2name Description of variable2.
 * ...
 * [Insert 2 spaces after the longest $variable definition, and then line
 * up all descriptions with this description]
```

```
*
* @return datatype Description of return value.
* @abstract [Only if necessary]
* @see Name of related Function/Method [if available]
* @since Akelos x.x [Only if necessary - use if function is added to the
* current release versions to indicate that the function has not been
* available in previous versions.]
*/
```

---

## Inline comments

When to use inline comments and when not to? These are some guidelines on what to do.

Inline comments are in general a means to explain what is going on in the code, however they have these problems.

- Hidden information. They are commonly hidden for developers who use the API of the system. For instance critical information on what is considered valid input is not publicly available.
- The code can get harder to read when comments and code are mixed.
- Comments are in many cases excuses for writing bad or hard to read code.

Things which should be considered before writing inline comments.

- Can the class or function be given a better name to reflect what it does?
- Can the variable names or the order of execution be changed to better reflect what is happening?
- Does the inline comment contain vital information that should be explained in the API (PHPDoc) doc?
- Can the code be simplified or split up to make it easier to read and understand.

In most cases the considerations above will help avoid inline comments. However in some cases they are required. Typical elements that require inline comments.

- Regular expressions, understanding what a RegExp does takes time. A comment will help here.
- Internal hacks, for instance due to performance problems or PHP specific issues.

An inline comment alone is not always sufficient, sometimes it can be wise to also include the same or similar text in the API documentation. For instance to give information that it uses a workaround on a specific PHP version.

Use of Perl/shell style comments (#) is discouraged.

All inline comments should be written using end-of-line comments //, like this.

```
// A workaround for PHP 4.2.2 were the xyz functionality is broken
if ( php_version() == '4.2.2' ){
    ...
}
```

# Error Return Check Policy

Check every system call for an error return, unless you know you wish to ignore errors.

Include the system error text for every system error message.

Akelos code should use PHP's [trigger\\_error\(\)](#) function for error reporting.

All Akelos code must work with `error_reporting = E_ALL`. Failure to do so would result in ugly output, error logs getting filled with lots of warning messages, or even downright broken scripts.

# Do Not do Real Work in Object Constructors

- Do not do any real work in an object's constructor. Inside a constructor initialize variables only and/or do only actions that can't fail.
- Create an Init() method for an object which completes construction. Init() should be called after object instantiation.

## Justification

- Constructors can't return an error.

```
class Device
{
    function Device()
    {
        /* initialize and other stuff */
    }
    function init()
    {
        ...
        return false;
    }
}

$Device =& new Device();

if (!$Device->init()){
    trigger_error('Unable to create Device Object', E_WARNING);
}
```

# Internationalization (I18n)

- All strings presented to the user **MUST** be translation-friendly. This is done by calling the `Ak::t()` function which takes care of translating the string to current selected language.
- ```
echo Ak::t("Hello world");
```

For strings that contain variable values, you can use the following syntax to replace any variable from the string

```
echo Ak::t("Hello %name. Today is %day_name.",
    array(
        'name'=>$name,
        'day_name'=>$Date->Today()
    )
);
```

This allows to simplify text translations.

[TODO: review `AkString` as it's not implemented on the framework yet]

- Don't use the `Ak::t()` function for strings that will be written to a log file or otherwise presented to the administrator.
- The `AkString::` class contains several string manipulation methods that are, as opposed to their PHP equivalents, locale and charset safe.
- Use `Ak::recode()` if you need to convert between different character set encodings (for example, between user input and a storage backend or data from an external source and the user interface). You don't need to care if the character sets are really different.
- Use the `AkString::lower()` and `AkString::upper()` methods without a second parameter if you need to perform a locale-independent string conversion. That's the case for all strings that are further processed or interpreted by code. Use these methods with the second parameter set to true for strings that need to be converted correctly according to the current (or specified) character set.
- Use the other `AkString::` equivalents of PHP string functions to manipulate strings correctly according to the current (or specified) character set but use the PHP functions for code/machine processed strings.

# Booleans and null

While PHP supports any case on the boolean types and the null type they must be written in lowercase.

```
$is_set = true;  
$can_print = false;  
$unset = null;
```

# Database Naming Conventions

All database tables used by Akelos resources and Akelos applications need to make sure that their table and field names work in all databases. Many databases reserve words like 'uid', 'user', etc. for internal use, and forbid words that are SQL keywords (select, where, etc.). Also, all names should be lowercase, with underscores ('\_') to separate words, to avoid case sensitivity issues.

A good way to do this for field names is to make the field name tablename\_fieldname.

Other general guidelines: Table names should be plural (users); field names should be singular (user\_name).

---

Here is a table with words that are reserved in databases supported by Akelos Framework:

|               |                   |              |                  |
|---------------|-------------------|--------------|------------------|
| ROWID         | ABSOLUTE          | ACCESS       | ACTION           |
| ADD           | ADMIN             | AFTER        | AGGREGATE        |
| ALIAS         | ALL               | ALLOCATE     | ALTER            |
| ANALYSE       | ANALYZE           | AND          | ANY              |
| ARE           | ARRAY             | AS           | ASC              |
| ASENSITIVE    | ASSERTION         | AT           | AUDIT            |
| AUTHORIZATION | AUTO_INCREMENT    | AVG          | BACKUP           |
| BDB           | BEFORE            | BEGIN        | BERKELEYDB       |
| BETWEEN       | BIGINT            | BINARY       | BIT              |
| BIT_LENGTH    | BLOB              | BOOLEAN      | BOTH             |
| BREADTH       | BREAK             | BROWSE       | BULK             |
| BY            | CALL              | CASCADE      | CASCADED         |
| CASE          | CAST              | CATALOG      | CHANGE           |
| CHAR          | CHAR_LENGTH       | CHARACTER    | CHARACTER_LENGTH |
| CHECK         | CHECKPOINT        | CLASS        | CLOB             |
| CLOSE         | CLUSTER           | CLUSTERED    | COALESCE         |
| COLLATE       | COLLATION         | COLUMN       | COLUMNS          |
| COMMENT       | COMMIT            | COMPLETION   | COMPRESS         |
| COMPUTE       | CONDITION         | CONNECT      | CONNECTION       |
| CONSTRAINT    | CONSTRAINTS       | CONSTRUCTOR  | CONTAINS         |
| CONTAINSTABLE | CONTINUE          | CONVERT      | CORRESPONDING    |
| COUNT         | CREATE            | CROSS        | CUBE             |
| CURRENT       | CURRENT_DATE      | CURRENT_PATH | CURRENT_ROLE     |
| CURRENT_TIME  | CURRENT_TIMESTAMP | CURRENT_USER | CURSOR           |
| CYCLE         | DATA              | DATABASE     | DATABASES        |
| DATE          | DAY               | DAY_HOUR     | DAY_MICROSECOND  |
| DAY_MINUTE    | DAY_SECOND        | DBCC         | DEALLOCATE       |
| DEC           | DECIMAL           | DECLARE      | DEFAULT          |
| DEFERRABLE    | DEFERRED          | DELAYED      | DELETE           |
| DENY          | DEPTH             | DEREF        | DESC             |
| DESCRIBE      | DESCRIPTOR        | DESTROY      | DESTRUCTOR       |
| DETERMINISTIC | DIAGNOSTICS       | DICTIONARY   | DISCONNECT       |
| DISK          | DISTINCT          | DISTINCTROW  | DISTRIBUTED      |
| DIV           | DO                | DOMAIN       | DOUBLE           |
| DROP          | DUMMY             | DUMP         | DYNAMIC          |
| EACH          | ELSE              | ELSEIF       | ENCLOSED         |
| END           | END-EXEC          | EQUALS       | ERRLVL           |
| ESCAPE        | ESCAPED           | EVERY        | EXCEPT           |

|                    |                    |                |                  |
|--------------------|--------------------|----------------|------------------|
| EXCEPTION          | EXCLUSIVE          | EXEC           | EXECUTE          |
| EXISTS             | EXIT               | EXPLAIN        | EXTERNAL         |
| EXTRACT            | FALSE              | FETCH          | FIELDS           |
| FILE               | FILLFACTOR         | FIRST          | FLOAT            |
| FOR                | FORCE              | FOREIGN        | FOUND            |
| FRAC_SECOND        | FREE               | FREETEXT       | FREETEXTTABLE    |
| FREEZE             | FROM               | FULL           | FULLTEXT         |
| FUNCTION           | GENERAL            | GET            | GLOB             |
| GLOBAL             | GO                 | GOTO           | GRANT            |
| GROUP              | GROUPING           | HAVING         | HIGH_PRIORITY    |
| HOLDLOCK           | HOST               | HOUR           | HOUR_MICROSECOND |
| HOURL_MINUTE       | HOURL_SECOND       | IDENTIFIED     | IDENTITY         |
| IDENTITY_INSERT    | IDENTITYCOL        | IF             | IGNORE           |
| ILIKE              | IMMEDIATE          | IN             | INCREMENT        |
| INDEX              | INDICATOR          | INFILE         | INITIAL          |
| INITIALIZE         | INITIALLY          | INNER          | INNODB           |
| INOUT              | INPUT              | INSENSITIVE    | INSERT           |
| INT                | INTEGER            | INTERSECT      | INTERVAL         |
| INTO               | IO_THREAD          | IS             | ISNULL           |
| ISOLATION          | ITERATE            | JOIN           | KEY              |
| KEYS               | KILL               | LANGUAGE       | LARGE            |
| LAST               | LATERAL            | LEADING        | LEAVE            |
| LEFT               | LESS               | LEVEL          | LIKE             |
| LIMIT              | LINENO             | LINES          | LOAD             |
| LOCAL              | LOCALTIME          | LOCALTIMESTAMP | LOCATOR          |
| LOCK               | LONG               | LOB            | LONGTEXT         |
| LOOP               | LOW_PRIORITY       | LOWER          | MAIN             |
| MAP                | MASTER_SERVER_ID   | MATCH          | MAX              |
| MAXEXTENTS         | MEDIUMBLOB         | MEDIUMINT      | MEDIUMTEXT       |
| MIDDLEINT          | MIN                | MINUS          | MINUTE           |
| MINUTE_MICROSECOND | MINUTE_SECOND      | MLSLABEL       | MOD              |
| MODE               | MODIFIES           | MODIFY         | MODULE           |
| MONTH              | NAMES              | NATIONAL       | NATURAL          |
| NCHAR              | NCLOB              | NEW            | NEXT             |
| NO                 | NO_WRITE_TO_BINLOG | NOAUDIT        | NOCHECK          |
| NOCOMPRESS         | NONCLUSTERED       | NONE           | NOT              |
| NOTNULL            | NOWAIT             | NULL           | NULLIF           |
| NUMBER             | NUMERIC            | OBJECT         | OCTET_LENGTH     |
| OF                 | OFF                | OFFLINE        | OFFSET           |
| OFFSETS            | OID                | OLD            | ON               |
| ONLINE             | ONLY               | OPEN           | OPENDATASOURCE   |
| OPENQUERY          | OPENROWSET         | OPENXML        | OPERATION        |
| OPTIMIZE           | OPTION             | OPTIONALLY     | OR               |
| ORDER              | ORDINALITY         | OUT            | OUTER            |
| OUTFILE            | OUTPUT             | OVER           | OVERLAPS         |
| PAD                | PARAMETER          | PARAMETERS     | PARTIAL          |
| PATH               | PCTFREE            | PERCENT        | PLACING          |
| PLAN               | POSITION           | POSTFIX        | PRECISION        |
| PREFIX             | PREORDER           | PREPARE        | PRESERVE         |
| PRIMARY            | PRINT              | PRIOR          | PRIVILEGES       |
| PROC               | PROCEDURE          | PUBLIC         | PURGE            |
| RAISERROR          | RAW                | READ           | READS            |
| READTEXT           | REAL               | RECONFIGURE    | RECURSIVE        |
| REF                | REFERENCES         | REFERENCING    | REGEXP           |
| RELATIVE           | RENAME             | REPEAT         | REPLACE          |
| REPLICATION        | REQUIRE            | RESOURCE       | RESTORE          |
| RESTRICT           | RESULT             | RETURN         | RETURNS          |
| REVOKE             | RIGHT              | RLIKE          | ROLE             |
| ROLLBACK           | ROLLUP             | ROUTINE        | ROW              |

|               |                     |                     |                  |
|---------------|---------------------|---------------------|------------------|
| ROWCOUNT      | ROWGUIDCOL          | ROWID               | ROWNUM           |
| ROWS          | RULE                | SAVE                | SAVEPOINT        |
| SCHEMA        | SCOPE               | SCROLL              | SEARCH           |
| SECOND        | SECOND_MICROSECOND  | SECTION             | SELECT           |
| SENSITIVE     | SEPARATOR           | SEQUENCE            | SESSION          |
| SESSION_USER  | SET                 | SETS                | SETUSER          |
| SHARE         | SHOW                | SHUTDOWN            | SIMILAR          |
| SIZE          | SMALLINT            | SOME                | SONAME           |
| SPACE         | SPATIAL             | SPECIFIC            | SPECIFICTYPE     |
| SQL           | SQL_BIG_RESULT      | SQL_CALC_FOUND_ROWS | SQL_SMALL_RESULT |
| SQL_TSI_DAY   | SQL_TSI_FRAC_SECOND | SQL_TSI_HOUR        | SQL_TSI_MINUTE   |
| SQL_TSI_MONTH | SQL_TSI_QUARTER     | SQL_TSI_SECOND      | SQL_TSI_WEEK     |
| SQL_TSI_YEAR  | SQLCODE             | SQLERROR            | SQLEXCEPTION     |
| SQLITE_MASTER | SQLITE_TEMP_MASTER  | SQLSTATE            | SQLWARNING       |
| SSL           | START               | STARTING            | STATE            |
| STATEMENT     | STATIC              | STATISTICS          | STRAIGHT_JOIN    |
| STRIPED       | STRUCTURE           | SUBSTRING           | SUCCESSFUL       |
| SUM           | SYNONYM             | SYSDATE             | SYSTEM_USER      |
| TABLE         | TABLES              | TEMPORARY           | TERMINATE        |
| TERMINATED    | TEXTSIZE            | THAN                | THEN             |
| TIME          | TIMESTAMP           | TIMESTAMPADD        | TIMESTAMPDIFF    |
| TIMEZONE_HOUR | TIMEZONE_MINUTE     | TINYBLOB            | TINYINT          |
| TINYTEXT      | TO                  | TOP                 | TRAILING         |
| TRAN          | TRANSACTION         | TRANSLATE           | TRANSLATION      |
| TREAT         | TRIGGER             | TRIM                | TRUE             |
| TRUNCATE      | TSEQUAL             | UID                 | UNDER            |
| UNDO          | UNION               | UNIQUE              | UNKNOWN          |
| UNLOCK        | UNNEST              | UNSIGNED            | UPDATE           |
| UPDATETEXT    | UPPER               | USAGE               | USE              |
| USER          | USER_RESOURCES      | USING               | UTC_DATE         |
| UTC_TIME      | UTC_TIMESTAMP       | VALIDATE            | VALUE            |
| VALUES        | VARBINARY           | VARCHAR             | VARCHAR2         |
| VARCHARACTER  | VARIABLE            | VARYING             | VERBOSE          |
| VIEW          | WAITFOR             | WHEN                | WHENEVER         |
| WHERE         | WHILE               | WITH                | WITHOUT          |
| WORK          | WRITE               | WRITETEXT           | XOR              |
| YEAR          | YEAR_MONTH          | ZEROFILL            | ZONE             |

# Example URLs

Use *example.com* for all example URLs, per [RFC 2606](#).

# Existence checking

Often you'll need to check whether or not a variable or property exists.

There are several cases here:

1. If you need to know if a variable exists at all and is not *null*, use [isset\(\)](#):

```
2. // Check to see if $param is defined.
3. if (isset($param)) {
4.     // $param may be false, but it's there.
5. }
```
6. If you need to know if a variable exists AND has a non-empty value (not *null*, 0, *false*, empty string or undefined), use [!empty\(\)](#):

```
7. // Make sure that $answer exists, is not an empty string, and is
8. // not 0:
9. if (!empty($answer)) {
10.     // $answer has some non-false content.
11. } else {
12.     // (bool)$answer would be false.
13. }
```

As pointed out in the comment of the else clause, [empty\(\)](#) essentially does the same check as [isset\(\)](#) -- is this variable defined in the current scope? -- and then, if it is, returns what the variable would evaluate to as a boolean. This means that 0, while potentially valid input, is "empty" - so if 0 is valid data for your case, don't use [!empty\(\)](#).

14. If you know you are working with a mixed variable then using just [isset\(\)](#) and [empty\(\)](#) could cause unexpected results, for example if testing for a key and the variable is actually a string:

```
15. $foo = 'bar';
16. if (isset($foo['somekey'])) {
17.     // This will evaluate to true
18. }
```

If you know that there is a possibility of a mixed type variable the solution in this case would be to add an [is\\_array\(\)](#) check in the *if()* statement.

19. Use [array\\_key\\_exists\(\)](#) when you want to check if an array key is defined even if it has a value of *null*:

```
20. // Make sure we have a charset parameter. Value could also be null.
21. if (!array_key_exists('charset', $params)) {
22.     trigger_error('Incomplete configuration.', E_WARNING);
23. }
```

Please note that [array\\_key\\_exists\(\)](#) is a performance hit (25%-100%) and should only be used when necessary. Instead try to use [!empty\(\)](#) or [isset\(\)](#) instead.

# php.ini settings

All Akelos code should work with [register\\_globals](#) disabled. This means using `$_COOKIE`, `$_SESSION`, `$_SERVER` and `$_ENV` to access all cookie, session, server and environment data, respectively.

To retrieve posted data (in the global `$_GET` and `$_POST` variables), you should normally use `Controller::params` which automatically references to `_GET` and `_POST` regardless of the setting of [magic\\_quotes\\_gpc](#).

All Akelos code should work with [error\\_reporting](#) = `E_ALL`. Failure to do so would result in ugly output, error logs getting filled with lots of warning messages, or even downright broken scripts.

No Akelos code should assume that `'.'` is in the include path. Always specify `'./'` in front of a filename when you are including a file in the same directory.

# Make Functions Reentrant

Functions should not keep static variables that prevent a function from being reentrant.

## Note

Reentrant function

A reentrant function is a function which can be invoked any number of times in parallel, without interference between the various invocations.

# Parameter Passing

Objects should be passed by reference. Everything else, including arrays, should be passed by value wherever semantically possible.

# Regular Expression Use

Always use the [preg\\_](#) functions if possible instead of `<a href="http://www.php.net/manual/en/ref.regex.php" ereg_` (and [preg\\_split\(\)](#) instead of [split\(\)](#)); they are included in PHP by default and much more efficient and much faster than [ereg\\_](#).

**NEVER** use a regular expression to match or replace a static string.

[explode\(\)](#) (in place of [split\(\)](#)), [str\\_replace\(\)](#), [strpos\(\)](#), or [strtr\(\)](#) do the job much more efficiently.

# XHTML 1.0 Compliance

All tag names and parameters must be lower case including javascript event handlers

```
<a href="http://example.com" onmouseover="status=''"
onmouseout="status=''">...</a>
```

All tag parameters must be of a valid parameter="value" form (numeric values must also be surrounded by quotes). For parameters that had no value in HTML, the parameter name is the value. For example:

```
<input type="checkbox" checked="checked" />
<select name="example">
  <option selected="selected" value="1">Example</option>
</select>
<td nowrap="nowrap">Example</td>
```

All tags must be properly closed. Tags where closing is forbidden must end with a space and a slash:

```
<br />
<hr />

<input type="submit" value="Example" />
```

All form definitions must be on their own line and either fully defined within a pair or be outside table tags. Forms must also always have an action parameter:

```
<form method="post" action="http://example.com/example.php">
<table>
  <tr><td>example</td></tr>
</table>
</form>

<table>
  <tr><td>
    <form action="javascript:void(0)" onsubmit="return false;">
    </form>
  </td></tr>
</table>
```

All JavaScript tags must have a valid language and type parameters:

```
<script language="JavaScript" type="text/javascript">
<!--
...
// -->
</script>
```

Nothing may appear after .

All images must have an alt attribute:

```
" />
```

Input fields of type "image" do not allow the border attribute and may render with a border on some browsers. Use the following instead:

```
<a href="" onclick="document.formname.submit(); return false;">"></a>
```