Introduction

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.

Reasons for a Standard

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 :-)

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 Do It then you should probably revisit your design.

Class Names

- The vast majority of classes you will program will be module classes. Modules should--in almost all cases--be derived from the names of the database tables they most directly rely on. For example: if a module is built to edit data in the "website_pages" table (refer ps_xento_dev database)--and displays information about multiple website pages, the module should be named CManagePsWebsitePages.class.php". If it only shows data that corresponds to one website page, it should be named CManagePsWebsitePage.class.php.
- If a class does not refer directly to a specific table, 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.
- When in doubt, look at the /Applications/Admin application layer for guidance. You should see correct naming syntax for modules classes.

Getting vs. Fetching

In order to distinguish between the retrieval of a member variable from memory and the execution of SQL to retrieve data from the database and load it into a member variable, functions are distinguished by "get" and "fetch".

All functions that result in a SQL query being executed on the database should have a name that begins with fetch. All functions that simply retrieve data already loaded into a member variable on an object should be preceded by get.

For example: if an object has a member variable called id = to 45 and you need to build a function to retrieve that data, the function would be getId(). If a CCompanyProperty object had corresponding CPhoto objects that needed to be retrieved from the database, a function called fetchPhotos() would be written.

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: getHtmlStatistics.
- Do not use: getHTMLStatistics.

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

```
class fluidOz             // NOT fluidOZ
class getHtmlStatistics      // NOT getHTMLStatistics
```

Class Names

- Use upper case letters as word separators, lower case for the rest of a word
- First character in a name is an upper case C
- No underbars ('_')

Example

```
class CManagePsWebsitePages
```

Current Conventions

Module classes in the different application layers should never be duplicated. To prevent this, use the following rules to name module classes in each application layer.

- Admin : CManageProperty.class.php
- PsWebsite : CDirectProperty.class.php

Zend/eclipse will have issues if you have duplicate file names in the same repository.

Class Attribute Names

Class member attribute names should be prepended with the characters 'm_'.

- After the 'm_' use the same rules as for class names.
- 'm_' always precedes other name modifiers like Hungarian notation.
- Class member variables for modules should be protected or private--never public or var.

- Do not initialize member variables in the declaration section, if needed it should be inside the class constructor.
- Do not initialize member variables with NULL as they are by default NULL.
- Declare access specifier of $_REQUEST_FORM_ variables as private static and accessing these variables using self::$ rather than $this->.

Justification

- Pre-pending 'm_' prevents any conflict with method argument names. Often your methods arguments and attribute names will be similar.
- Initializing member variables in declaration section will reserve memory space at each instance. Better to initialize them where they are used.
- We don't need to initialize variables with NULL as they are by default NULL.
- 'static' keyword allocates memory once, so we can avoid extra memory creation on every object called.

Example

protected $m_strVarAbc;

Method Argument Names

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

Justification

- You can always tell which variables are passed in methods.

Example

```
class CNameOneTwo {

    protected $m_strSomeEngine;
    protected $m_strAnotherEngine;

    function startYourEngines( $strSomeEngine, $strAnotherEngine ) {
        $this->m_strSomeEngine = $strSomeEngine;
        $this->m_strAnotherEngine = $strAnotherEngine;
    }
}
```

Variable Names

Hungarian Notation

- All variable and class member names should be prepended using a modified Hungarian Notation.
- Arrays use a compound notation.

Allowed prefix

- Simple variables:

```
int  // Integer
flt  // Float
str  // String
```

```
obj  // Object
bool // Boolean
mix  // Mixed
res  // Resource
fun // Callable function
```

- Array variables:

```
arrint  // Array of Integer
arrflt  // Array of Float
arrstr  // Array of Strings
arrobj  // Array of Object
arrbool // Array of Boolean
arrmix  // Array of Mixed
arrres  // Array of Resource
arrfun // Array of callables
```

- Special variable

```
$_REQUEST_FORM_
```

Examples

Wrong Usage	Correct usage
$floatRentChargeAmount	$fltRentChargeAmount

Array Element

Except where noted, array element names follow the same rules as a variable.

- array element names should be all lowercase
- use '_' as the word separator.
- don't use '-' as the word separator
- use single quotes when defining array elements

Justification

- if used with '-' as the word separator, the array's element can be accessed by magic quotes.

Example

```
$arrstrMyArr['foo_bar'] = 'Hello';
$arrstrMyArr['foo-bar'] = 'Hello';
$arrstrMyArr[foo_bar] = 'Hello'; //undefined constant error
print "$arrstrMyArr[foo_bar] world"; // will output: Hello world
print "$arrstrMyArr[foo-bar] world"; // parse error
```

Define Names / Global Constants

- Global constants should be all caps with '_' separators.

Justification

It's tradition for global constants to named this way. You must be careful to not conflict with other predefined globals.

Constants should be defined as class constants wherever possible--as opposed to global defines.

Example

```
define( 'A_GLOBAL_CONSTANT', 'Hello world!' );
const A_GLOBAL_CONSTANT = 'Hellow World!';
```

Static Variables

- Static variables may be prepended with 'c_'.

Justification

- It's important to know the scope of a variable.

Example

```
function test() {
  static $c_intStatus = 0;
}
```

Function Names

- Use the same approach as naming method names.

No Variable Variables

- Though php allows the use of variable variables, it leads to unreadable code.

Example

```
$varA = 'getAmount';
$intAmount = {$varA}; // Do not use
$intAmount = $$varA;  // Do not use
```

No Variable Methods

- Same rules apply as No Variable Variables

Common Folder Name

- Folder containing common files should be named as _common

Justification

- This will make the common folder standout
- It will show at the top of the list of folders

Appropriate Database Object Names

While writing new functions do mention appropriate database object type. e.g.

If the function fetches data from client database (works schema), then use variable $objClientDatabase, and use $objMasterDatabase for fetching data from master database.

- Correct Use

public function fetchPropertiesByCompanyAccountId($intCompanyAccountId, $objMasterDatabase) {}

function fetchPropertiesByCustomerId($intCustomerId, $objClientDatabase){}

- Not to be used

function fetchPropertiesByCustomerId($intCustomerId, $objDatabase){}

Basic Naming Conventions

- a) Member variables:

Class member attribute names should be prepended with the characters 'm_'.
After the 'm_' use the same rules as for class names.
Class member variables for modules should be protected or private--**never public or var**.

- b) Function Name:

The first character should be lowercase followed by logical name starting with uppercase character.

- c) Method Argument Names:

The first character should be lowercase.
All word beginnings after the first letter should be uppercase as with class names.

- d) Variable Names:

All variable and class member names should be prepended using a modified Hungarian Notation
(Hungarian notation is an identifier naming convention in computer programming, in which the name of a variable or function indicates its type or intended use.)

```
Allowed prefix -
   Simple variables:
      int // Integer
      flt // Float
      str // String
      obj // Object
      bool // Boolean
      mix // Mixed
      res // Resource

  Array variables:
      arrint // Array of Integer
      arrflt // Array of Float
      arrstr // Array of Strings
```

```
    arrobj // Array of Object
    arrbool // Array of Boolean
    arrmix // Array of Mixed
    arrres // Array of Resource
    For e.g $strBrowserName
```

- e) Static Variables

```
   Static variables may be prepended with 'c_' as it belongs to class.
    For e.g
    function test() {
    static $c_intStatus = 0;
    }
```

Accessing Variables

- Private and Protected members should only be accessed using the members get/set methods (if they exist) from non related classes.
- Private and Protected members should usually be accessed using the members get/set methods (if they exist) from related classes.
- Public members should usually be accessed using the members get/set methods (if they exist) from non related classes.

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.

Example

```
$f = @fopen( ... );
```

Braces {} Policy

Of the three major brace placement strategies only the Traditional Unix policy is acceptable:

- 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:

Example

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

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

Indentation/Tabs/Space Policy

- Indent using size 4 tabs
- Do not use spaces, use 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.
- Indent array literals the same as you would indent control structures: once per level.

Justification

- When people using different tab settings the code is impossible to read or print, which is why everybody needs to have the same tab size.
- Most PHP applications use size 4 tabs.
- Most editors use size 4 tabs by default
- 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.
- Lines of code that require scrolling horizontally in order to see the whole line are very hard to quickly read and understand. Indenting the minimum amount necessary will help avoid this.

Examples

```
function func() {
    if ( something bad ) {
        if ( another thing bad ) {
            while ( more input ) {
            }
        }
    }
}
```

With array literals, don't do this:

```
$arrmixComplicatedData = array(
                    'some_key'    => array(
                                     'some_nested_key' => array(
                                                      'doubly_nested_key' => 0
                                                     )
                                   ),
                    'some_other_key' => array()
                   );
```

One level of indentation is enough for us to understand the hierarchy of our control structures, so it should also be enough to understand the hierarchy of an array structure. Even though the above may have some pleasing aspects to the way things line up, code like this will quickly lead to unnecessary horizontal scrolling, and make it harder to read and understand.

Do this instead:

```
$arrmixComplicatedData = array(
    'some_key' => array(
        'some_nested_key' => array(
            'doubly_nested_key' => 0
        )
    ),
    'some_other_key' => array()
);
```

Parens () with Key Words and Functions Policy

- Do not put parens next to parameters. Put a space between.
- Do put parens next to keywords.
- Do put parens next to function names.
- Do not use parens in return statements when it's not necessary.

Justification

- Keywords are not functions. By putting parens next to keywords and function names are made to look alike.

Example

```
if( condition ) {
}

while( condition ) {
}

strcmp( $s, $s1 );

return 1;
```

Line breaks

Logical blocks or control structures like if, for, while, etc should have a blank line before and after the control block for logical separation. It improves the code readability.

Variable blocks

- Note the line breaks after collection of variables
- Note the variables are aligned based on variable name length
- Note the clubbing of same data type variables

```
protected $m_objClient;
protected $m_objFormFilter;
protected $m_objPagination;
protected $m_objCompanyUser;

protected $m_objDatabase;
protected $m_objSmsDatabase;
protected $m_objChatDatabase;
protected $m_objPricingDatabase;
protected $m_objScreeningDatabase;
protected $m_objInsurancePortalDatabase;
```

```
        protected $m_boolIsDryRun           = false;
        protected $m_boolIsSchedulable             = false;
        protected $m_boolDisplaySearchFilter       = true;
        protected $m_boolIsDisplayOptionsVisible = true;
```

Logical blocks

- Note:
    - Line breaks before & after try-catch block
    - Line breaks before & after $arrmixReportData assignments
    - Line breaks before & after if() block

```
public function viewReportData( $strOutputType, $strQueryParams = '' ) {

        try {
                $arrmixReportData = $this->viewReport( $strOutputType );
        } catch( Exception $objException ) {
                return ['error' => $this->m_arrstrErrors];
        }

        $arrmixReportData['report_object']       = $this->getReportObject();
        $arrmixReportData['report_name']         = $this->getReportName();
        $arrmixReportData['report_details']      = $this->viewReportDetails();
        $arrmixReportData['report_identifier']   = $this->getReportIdentifier();
        $arrmixReportData['module']              = $this->getModuleName();

        if( true == isset( $this->m_objUserTime ) ) {
                $arrmixReportData['user_time'] = $this->m_objUserTime;
        }

        if( 'screen' == $strOutputType ) {
                if( true == $this->getIsPagination() ) {
                        $arrmixReportData['pagination_url'] = '/?module=' .
$this->getModuleName() . '&action=view_report&is_library_report=1' . $strQueryParams;
                }

                $arrmixReportData['display_pagination']    = $this->getIsPagination();
                $arrmixReportData['display_search_filter'] = $this->getDisplaySearchFilter();
        }

        return $arrmixReportData;
}
```

Control structures

```
if( condition ) {
  statements;
}

while( condition ) {
  statements;
}
```

Function blocks

```
public function getDisplaySearchFilter() {
        return $this->m_boolDisplaySearchFilter;
}

>m_objDatabase;
}
```

Examples

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 Open() method for an object which completes construction. Open() should be called after object instantiation.

Justification

- Constructors can't return an error.

Example

```
class Device {
   function device() { /* initialize and other stuff */ }
   function open() { return FAIL; }
}

$dev = new Device;
if( FAIL == $dev->Open() ) exit( 1 );
```

Make Functions Reentrant

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

If Then Else Formatting

Layout

Different bracing styles will yield slightly different looks. The approach we will use is:

```
if( condition ) {              // Comment
} elseif( condition ) {        // Comment
} else {                       // Comment
}
```

If you have *elseif* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

Condition Format

Always put the constant on the left hand side of an equality/inequality comparison. For example:

```
if( 6 == $errorNum ) ...
```

One reason is that if you leave out one of the = signs, the parser will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

Switch Formatting

- Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
- 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.

Example

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

   // FALL THROUGH
   case 2:
      $v = get_week_number();
      ...
      break;
   default:
}
```

Alignment of Declaration Blocks

- Block of declarations should be aligned.

Justification

- Clarity.
- Similarly blocks of initialization of variables should be tabulated.
- The ‘&’ token should be adjacent to the type, not the name.

Example

```
var      $m_intDate
var&      $m_resDate
var&      $m_resName
var      $m_strName

$m_intDate    = 0;
$m_resDate    = NULL;
$m_resName    = 0;
$m_strName    = NULL;
```