Contents

[Introduction 5](#_Toc373696797)

[Important Things 5](#_Toc373696798)

[General Coding Conventions 6](#_Toc373696799)

[**1.** **Naming** 6](#_Toc373696800)

[**1.1.** **Class Names** 6](#_Toc373696801)

[**1.2.** **Interfaces** 6](#_Toc373696802)

[**1.3.** **Function/Method Names** 6](#_Toc373696803)

[**1.4.** **Getters/Setters** 7](#_Toc373696804)

[**1.5.** **Variables** 7](#_Toc373696805)

[**1.6.** **Global Variables** 7](#_Toc373696806)

[**1.7.** **Constants** 7](#_Toc373696807)

[**2.** **File Naming** 8](#_Toc373696808)

[**2.1.** **Class Files** 8](#_Toc373696809)

[**2.2.** **Code File Names** 8](#_Toc373696810)

[**2.3.** **Image File Names** 8](#_Toc373696811)

[**3.** **Declarations** 8](#_Toc373696812)

[**3.1.** **Number Per Line** 9](#_Toc373696813)

[**3.2.** **Initialization** 9](#_Toc373696814)

[**3.3.** **Placement** 9](#_Toc373696815)

[**4.** **Prefixes** 10](#_Toc373696816)

[**5.** **Code Formatting** 12](#_Toc373696817)

[**5.1.** **Curly Brackets** 12](#_Toc373696818)

[**5.2.** **Blank Lines** 12](#_Toc373696819)

[**6.** **Comments** 12](#_Toc373696820)

[**6.1.** **Tag references** 13](#_Toc373696821)

[**6.2.** **File Comment** 16](#_Toc373696822)

[**6.3.** **Simple Single Line comments** 16](#_Toc373696823)

[**6.4.** **Function/Method Comments** 16](#_Toc373696824)

[**6.5.** **Variable, Constant, Attribute and other declarations’ Comment** 17](#_Toc373696825)

[**7.** **Indentation** 17](#_Toc373696826)

[**8.** **Variables** 17](#_Toc373696827)

[**9.** **Semi-Colons** 18](#_Toc373696828)

[**10.** **Statements** 18](#_Toc373696829)

[**10.1.** **Simple Statements** 18](#_Toc373696830)

[**10.2.** **Compound Statements** 18](#_Toc373696831)

[**10.3.** **Return Statements** 19](#_Toc373696832)

[**10.4.** **If, if-else, if else-if else Statements** 19](#_Toc373696833)

[**10.5.** **For Statements** 20](#_Toc373696834)

[**10.6.** **While Statements** 20](#_Toc373696835)

[**10.7.** **Do-while Statements** 21](#_Toc373696836)

[**10.8.** **Switch Statements** 21](#_Toc373696837)

[**10.9.** **Try-catch Statements** 22](#_Toc373696838)

[**11.** **Best Practices** 22](#_Toc373696839)

[**11.1.** **Readability of Code Blocks** 22](#_Toc373696840)

[**11.2.** **Return Early** 23](#_Toc373696841)

[**11.3.** **Variable Assignments** 24](#_Toc373696842)

[**11.4.** **Parentheses** 25](#_Toc373696843)

[**11.5.** **Returning Values** 25](#_Toc373696844)

[**11.6.** **Expression before ‘?’ in the conditional operator** 26](#_Toc373696845)

[HTML & CSS 27](#_Toc373696846)

[**1.** **Naming Conventions** 27](#_Toc373696847)

[**1.1.** **HTML & CSS File Names** 27](#_Toc373696848)

[**1.2.** **ID and Class Names** 27](#_Toc373696849)

[**1.3.** **ID and Class Name Delimiters** 28](#_Toc373696850)

[**2.** **Formatting** 28](#_Toc373696851)

[**3.** **Including Meta Data** 29](#_Toc373696852)

[**4.** **Targeting Internet Explorer** 29](#_Toc373696853)

[**5.** **HTML Validity** 30](#_Toc373696854)

[**6.** **CSS Validity** 31](#_Toc373696855)

[**7.** **Shorthand Properties** 31](#_Toc373696856)

[**8.** **Prefixes** 31](#_Toc373696857)

[**9.** **Selector and Declaration Separation** 32](#_Toc373696858)

[**10.** **Section Comments** 32](#_Toc373696859)

[JavaScript 34](#_Toc373696860)

[**1.** **Naming** 34](#_Toc373696861)

[**1.1.** **File Names** 34](#_Toc373696862)

[**2.** **Use JavaScript files** 34](#_Toc373696863)

[**3.** **Variables** 34](#_Toc373696864)

[**4.** **Constants** 35](#_Toc373696865)

[**5.** **Functions** 35](#_Toc373696866)

[**6.** **Statements** 37](#_Toc373696867)

[**6.1.** **Continue Statements** 38](#_Toc373696868)

[**6.2.** **With Statements** 38](#_Toc373696869)

[**7.** **Semi-Colons** 38](#_Toc373696870)

[**8.** **Function declaration within blocks** 40](#_Toc373696871)

[**9.** **Some Do’s** 40](#_Toc373696872)

[**9.1.** **Custom Exceptions** 40](#_Toc373696873)

[**9.2.** **Use Standard Features** 41](#_Toc373696874)

[PHP 42](#_Toc373696875)

[**1.** **Naming** 42](#_Toc373696876)

[**2.** **File Naming** 42](#_Toc373696877)

[**2.1.** **Template Files** 42](#_Toc373696878)

[**2.2.** **PHP Files** 42](#_Toc373696879)

[**2.3.** **Include Files** 42](#_Toc373696880)

[**3.** **PHP Code Demarcation** 43](#_Toc373696881)

[**4.** **String Literals** 43](#_Toc373696882)

[**5.** **Variable Substitution** 43](#_Toc373696883)

[**6.** **Arrays** 44](#_Toc373696884)

[**6.1.** **Numerically Indexed Arrays** 44](#_Toc373696885)

[**6.2.** **Associative Arrays** 44](#_Toc373696886)

[**7.** **Classes, Functions and Methods.** 45](#_Toc373696887)

[**8.** **Control Statements** 45](#_Toc373696888)

[Databases 46](#_Toc373696889)

[**1.** **General Naming Convention** 46](#_Toc373696890)

[**2.** **Database Users** 46](#_Toc373696891)

[**3.** **Database Name** 46](#_Toc373696892)

[**4.** **Table Name** 46](#_Toc373696893)

[**5.** **Column Name** 47](#_Toc373696894)

# Introduction

The long-term value of software is in direct proportion to the quality of the codebase. Over its lifetime, a program will be handled by many pairs of hands and eyes. If a program is able to clearly communicate its structure and characteristics, it is less likely that it will break when modified in the never-too-distant future. Code conventions can help in reducing the brittleness of programs.

In this document we discuss the various coding conventions that are mandatory for all Accrete Globus Technology Projects.

# Important Things

1. **Be Consistent**

If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around all their arithmetic operators, you should too. If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you're saying rather than on how you're saying it. We present global style rules here so people know the vocabulary, but local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Avoid this.

# General Coding Conventions

1. **Naming**

Naming is extremely important in every single line of code we write. We use some simple and general conventions for naming:

* Use names that are as short as possible but as long as necessary.
* Try to convey what a class, object, variable, or programming construct is about while being as brief as possible.

Using names this way contributes to acceptable levels of understandability and code efficiency. These are the basic guidelines to good naming conventions. Naming conventions are however discussed more in depth in the different sections of this document.

Names should be formed from the 26 upper and lower case letters (A .. Z, a .. z), the 10 digits (0 .. 9), and \_ (underscore). Avoid use of international characters because they may not read well or be understood everywhere. Do not use $ (dollar sign) or \ (backslash) in names.

Do not use \_ (underscore) as the first character of a name. It is sometimes used to indicate privacy, but it does not actually provide privacy. If privacy is important, use the forms that provide [private members](http://javascript.crockford.com/private.html). Avoid conventions that demonstrate a lack of competence.

Most variables and functions should start with a lower case letter.

Constructor functions which must be used with the [new prefix](http://yuiblog.com/blog/2006/11/13/javascript-we-hardly-new-ya/) should start with a capital letter.

**Global variables** should be in all caps.

* 1. **Class Names**

All class names should be camel case with the first letter being capital. Examples

class Person

class Computer

class DellComputer

* 1. **Interfaces**

Interfaces follow the same conventions as classes.

* 1. **Function/Method Names**

Function and Method Names are camel case names with the first letter being a common letter. Examples:

/\*\* @desc Java style method \*/

public String getFullName(){}

/\*\* @desc PHP style function/method \*/

public function organizeCategories(){}

* 1. **Getters/Setters**

Getters and setters methods for properties are not required, only create them if needed. However, if they are used, then getters must be named getFoo() and setters must be named setFoo(value). For boolean getters, isFoo() is recommended, as it often sounds more natural.

* 1. **Variables**

Variables are camel case names with the first letter being a common letter. Examples

var myName = “Joshua Kissoon”; // Javascript Style

$myName = “Joshua Kissoon”; // PHP Style

public String myName = “Joshua Kissoon”; // Java style

* 1. **Global Variables**

Constant names should always be in all caps.

* 1. **Constants**

Constant names should always be in all caps. Examples:

private static final String SOME\_CONST = “riu”; // Java Constant

define(“A\_PHP\_CONSTANT”, “some value”); // PHP Constant

1. **File Naming**

File Naming conventions are very important and should be constant throughout applications. Here we discuss general file naming conventions. However, under further sections, where the naming conventions may differ, it is stated there (as with HTML and CSS file names).

* 1. **Class Files**

Class file names should be the same as the class it contains and does not have any prefix or suffix.

/\* Not Recommended \*/

Person.class.java

/\* Recommended \*/

Person.java

FootballPlayer.php

* 1. **Code File Names**

File names should be all lower case letters and multi-word file names should be separated by underscores.

/\* Not Recommended \*/

Index.php

CustomerManagement.php

Photo-Gallery.js

Customer\_management.php

/\* Recommended \*/

index.php

customer\_management.php

photo\_gallery.js

* 1. **Image File Names**

Image file names should only include the letters A-Z, a-z, the numbers 0-9 and the characters \_ and -.

1. **Declarations**
   1. **Number Per Line**

One declaration per line is recommended since it encourages commenting. In other words,

int level; // indentation level

int size; // size of table

is preferred over

int level, size;

Do not put different types on the same line. Example:

/\* This is wrong \*/

int foo, fooarray[];

**Note:** The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

int level; // indentation level

int size; // size of table

Object currentEntry; // currently selected table entry

* 1. **Initialization**

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

* 1. **Placement**

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

void myMethod()

{

int int1 = 0; // beginning of method block

if (condition)

{

int int2 = 0; // beginning of "if" block

...

}

}

The one exception to the rule is indexes of for loops, which in most languages can be declared in the for statement:

for (int i = 0; i < maxLoops; i++)

{

...

}

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

int count;

...

myMethod()

{

if (condition)

{

int count = 0; // Avoid this!

...

}

...

}

1. **Prefixes**

Prefixes are especially important in large projects where there are many modules in the project; we need to separate the modules files from each other since they may get mixed up. Prefixes are also important for code that gets embedded in other projects or on external sites. Use short, unique identifiers. Using namespaces helps preventing naming conflicts and can make maintenance easier, for example in search and replace operations.

**Example:** Think of an application for a shopping website that needs to keep information about the companies it uses to ship items to its customers, companies supplies its items as well as partner companies. Each of these companies is managed by a different module in the system and their physical files (agreement documents, photos, payment receipts, etc) are all stored on the same file server. Think of how confusing it would be if we named everything company. The simple solution would be to use prefixes for the different types of companies.

/\* Module Names \*/

shipping\_company\_mod

supplier\_company\_mod

partner\_company\_mod

/\* File System Structure \*/

site/files/companies/

shipping

partner

supplier

/\* Database Table Names \*/

shipping\_company

shipping\_company\_manager

shipping\_company\_website

supplier\_company

supplier\_company\_address

supplier\_company\_website

partner\_company

partner\_company\_address

partner\_company\_manager

/\* Note: database naming conventions are discussed later in this document \*/

As we can see from above, making use of prefixes have solved the problems.

1. **Code Formatting**
   1. **Curly Brackets**

Curly Brackets (“{“ and “}”) specifying code blocks should always be in new lines; no code should be on the same line as the curly bracket. The opening and closing curly braces should align with each other. Examples:

P

{

font-size: 20px;

}

function get\_author($author\_id)

{

/\* some code that gets the author ID \*/

}

* 1. **Blank Lines**

Use newlines to group logically related pieces of code. For example:

doSomethingTo(x);

doSomethingElseTo(x);

andThen(x);

nowDoSomethingWith(y);

andNowWith(z);

1. **Comments**

Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly yourself) who will need to understand what you have done. The comments should be well-written and clear, just like the code they are annotating. An occasional nugget of humor might be appreciated. Frustrations and resentments will not.

It is important that comments be kept up-to-date. Erroneous comments can make programs even harder to read and understand.

* 1. **Tag references**

For block comments, like those used to describe a class or function/method (Examples below), we use tag references to allow for automated documenters to document a system. Example usage is shown below. Tag references used include:

* [*@desc*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/api.html): The @desc tag is used to specify the description of the Structural Element (Function, Method, Class, Object, etc).
* [*@api*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/api.html): The @api tag is used to declare Structural Elements as being suitable for consumption by third parties or other modules within a system.

/\*\*

\* @api Some API description can go here.

\* @desc Some Description

\*/

function showVersion()

{

<...>

}

* [*@author*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/author.html)
* [*@date*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/author.html)
* [*@deprecated*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/deprecated.html): The @deprecated tag is used to indicate which Structural elements are deprecated and are to be removed in a future version.

/\*\*

\* @deprecated

\* @deprecated 1.0.0

\* @deprecated No longer used by internal code and not recommended.

\* @deprecated 1.0.0 No longer used by internal code and not recommended.

\*/

function count()

{

<...>

}

* [*@example*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/example.html): Specifies the example file which contains an example of use of this Structure. Example files are stored in the example/ directory of any project.

/\*\*

\* @example example1.php Counting in action.

\* @example ex.com/example2.php Counting in action by a 3rd party.

\* @example "My Own Example.php" My counting.

\*/

function count()

{

<...>

}

* [*@internal*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/internal.html): The @internal tag is used to denote that associated [Structural Elements](http://www.phpdoc.org/docs/latest/for-users/introduction/definitions.html#term-structural-elements) are elements internal to this application or library. It may also be used inside a long description to insert a piece of text that is only applicable for the developers of this software. The same as a private method within a class.

/\*\*

\* @internal

\*/

function count()

{

<...>

}

* [*@param*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/param.html): The @param tag is used to document a single argument of a function or method.

/\*\*

\* @desc Counts the number of items in the provided array.

\* @param mixed[] $items Array structure to count the elements of.

\* @return int Returns the number of elements.

\*/

function count(array $items)

{

<...>

}

* [*@return*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/return.html): The @return tag is used to document the return value of functions or methods. Example is shown above.
* [*@throws*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/throws.html): The @throws tag is used to indicate whether [Structural Elements](http://www.phpdoc.org/docs/latest/for-users/introduction/definitions.html#term-structural-elements)could throw a specific type of exception.

/\*\*

\* @desc Counts the number of items in the provided array.

\* @param mixed[] $array Array structure to count the elements of.

\* @throws InvalidArgumentException if the provided argument is not of type 'array'.

\* @return int Returns the number of elements.

\*/

function count($items)

{

<...>

}

* [*@todo*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/todo.html): The @todo tag is used to indicate any outstanding development activities to be executed on associated [Structural Elements](http://www.phpdoc.org/docs/latest/for-users/introduction/definitions.html#term-structural-elements).

/\*\*

\* @desc Counts the number of items in the provided array.

\* @todo add an array parameter to count

\* @return int Returns the number of elements.

\*/

function count()

{

<...>

}

* [*@version*](http://www.phpdoc.org/docs/latest/for-users/phpdoc/tags/version.html): indicates the current version of [Structural Elements](http://www.phpdoc.org/docs/latest/for-users/introduction/definitions.html#term-structural-elements). Mostly used for classes or other major collections or representations.

/\*\*

\* @version 1.0.1

\*/

class Counter()

{

<...>

}

* 1. **File Comment**

Every file (code file, class file, etc) that contains code must have a docblock at the top of the file that contains these tags at a minimum:

/\*\*

\* @summary Short description for file

\* @desc Long description for file (if any)...

\* @package    Zend\_Magic

\* @copyright  Copyright (c) 2005-2013 AGT Inc(http://www.accreteglobal.com)

\* @version    $Id:$

\* @date

\* @author

\*/

* 1. **Simple Single Line comments**

Simple single line comments that describe a block of code should be bounded by /\*\*/. Example:

function get\_author\_name($author\_id)

{

/\* Load the author information for the given author id\*/

$result = $DB->query(…..);

$author = $DB->fetchObject($result);

/\* Now we return the author’s full name \*/

Return $author->first\_name + “ ” + $author->last\_name;

}

* 1. **Function/Method Comments**

The comments for a function or a class method should be a block style comment located above the function/method. The comment **should** include information about the parameters (@param), the return value of the function/method if any (@return) and a description (@desc) of the function/method.

/\*\*

\* @desc Gets the number of publications for an author for a year.

\* @param $author\_id Author’s unique ID

\* @param $year Which year’s publications are we counting.

\* @return The number of publications of an author for a specific year

\*/

function get\_author\_publications\_count ($author\_id, $year)

{

/\* Load the author information for the given author id for this year \*/

$result = $DB->query(…..);

$author = $DB->fetchObject($result);

/\* Now we return the author’s publication count \*/

Return $author->publications\_count;

}

* 1. **Variable, Constant, Attribute and other declarations’ Comment**

For declarations, we may want/need to write a one line comment to describe this variable/attribute/constant/etc. For these types of comments, use the // commenting style. Put the comment next to the declaration.

var jsVar; // A Javascript Variable

$name = “Joshua Kissoon”; // A PhP Variable

1. **Indentation**

Indent by four (4) spaces at a time. Don’t use tabs or mix tabs and spaces for indentation. Keep your indentation consistent throughout all of your code

<ul>

<li>Fantastic

<li>Great

</ul>

.example

{

color: blue;

}

1. **Variables**

All variables should be declared before used. Some languages does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied [globals](http://yuiblog.com/blog/2006/06/01/global-domination/). Implied global variables should never be used.

It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order.

var currentEntry; // currently selected table entry

var level; // indentation level

var size; // size of table

Use of global variables should be minimized. Implied global variables should never be used.

1. **Semi-Colons**

Always use semi-colons. Relying on implicit insertion can cause subtle, hard to debug problems. Don't do it. You're better than that.

1. **Statements**
   1. **Simple Statements**

Each line should contain at most one statement. Example:

argv++; // Correct

argc--; // Correct

argv++; argc--; // AVOID!

* 1. **Compound Statements**

Compound statements are statements that contain lists of statements enclosed in braces "{ statements }". See the following sections for examples.

* The enclosed statements should be indented one more level than the compound statement.
* The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
* Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.
  1. **Return Statements**

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

return;

return myDisk.size();

return (size ? size : defaultSize);

* 1. **If, if-else, if else-if else Statements**

The if-else class of statements should have the following form:

if (condition)

{

statements;

}

if (condition)

{

statements;

}

else

{

statements;

}

if (condition)

{

statements;

}

else if (condition)

{

statements;

}

Else

{

statements;

}

**Note:** if statements always use braces, {}. Avoid the following error-prone form:

if (condition) //AVOID! THIS OMITS THE BRACES {}!

statement;

* 1. **For Statements**

A for statement should have the following form:

for (initialization; condition; update)

{

statements;

}

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

for (initialization; condition; update);

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

* 1. **While Statements**

A while statement should have the following form:

while (condition)

{

statements;

}

An empty while statement should have the following form:

while (condition);

* 1. **Do-while Statements**

A do-while statement should have the following form:

do

{

statements;

} while (condition);

* 1. **Switch Statements**

A switch statement should have the following form:

switch (condition)

{

case ABC:

statements;

/\* falls through \*/

case DEF:

statements;

break;

case XYZ:

statements;

break;

default:

statements;

break;

}

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the /\* falls through \*/ comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

* 1. **Try-catch Statements**

A try-catch statement should have the following format:

try

{

statements;

}

catch (ExceptionClass e)

{

statements;

}

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

try

{

statements;

} catch (ExceptionClass e)

{

statements;

}

finally

{

statements;

}

1. **Best Practices**

Certain things are not mandatory but rather best practices; here we outline some of these.

* 1. **Readability of Code Blocks**

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

if ($foo)

{  
    $bar = 1;  
}  
if ($spam)

{  
    $ham = 1;  
}  
if ($pinky)

{  
    $brain = 1;  
}

is a lot easier to read when separated:

if ($foo)

{  
    $bar = 1;  
}  
  
if ($spam)

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

{  
    $brain = 1;  
}

* 1. **Return Early**

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

function foo($bar, $baz)  
{  
    if ($foo) {  
        //assume that here is  
        //the whole  
        //logic of  
        //this method  
        return $calculated\_value;  
    } else {  
        return null;  
    }  
}

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

function foo($bar, $baz)  
{  
    if (!$foo) {  
        return null;  
    }  
  
    //assume that here is  
    //the whole  
    //logic of  
    //this method  
    return $calculated\_value;  
}

* 1. **Variable Assignments**

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

fooBar.fChar = barFoo.lchar = 'c'; // AVOID!

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

if (c++ = d++) // AVOID! (Java disallows)

{

...

}

should be written as

if ((c++ = d++) != 0)

{

...

}

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

/\* Avoid \*/

d = (a = b + c) + r;

should be written as

a = b + c;

d = a + r;

### 

* 1. **Parentheses**

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

/\* Avoid this \*/

if (a == b && c == d)

/\* This is better \*/

if ((a == b) && (c == d))

#### 

* 1. **Returning Values**

Try to make the structure of your program match the intent. Example:

if ( booleanExpression)

{

return true;

}

else

{

return false;

}

/\* should instead be written as \*/

return booleanExpression;

Similarly,

if (condition)

{

return x;

}

return y;

/\* should be written as \*/

return (condition ? x : y);

#### 

* 1. **Expression before ‘?’ in the conditional operator**

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

/\* Correct way \*/

(x >= 0) ? x : -x;

# HTML & CSS

1. **Naming Conventions**
   1. **HTML & CSS File Names**

* All HTML and CSS file names should be only lowercase letters and numbers.
* HTML file names should be as short as possible but enough to describe what the file contain.
* HTML File names should also use prefixes where possible to depict which sub-section of templates a file belongs to; for example, login.html is a form and should thus be named form-login.html.
* HTML File names that contain multiple words should be separated by a hyphen.
* CSS file names should reflect which part of the site the CSS file styles
* Some General CSS files:
  + style.css: contains style general to the entire site.
  + reset.css: contain styles that helps make the website cross browser compatible.
* Some General HTML files:
  + main.html: contains the main html template file for the website which other components of the website are put into.

/\* Not Recommended \*/

User-signup.html

user\_signup.html

login.html

something.css

joshua.css

/\* Recommended \*/

form-user-signup.html

form-login.html

photo-gallery.css

* 1. **ID and Class Names**

Use meaningful or generic ID and class names: Instead of presentational or cryptic names, always use ID and class names that reflect the purpose of the element in question, or that are otherwise generic. Names that are specific and reflect the purpose of the element should be preferred as these are most understandable and the least likely to change. Generic names are simply a fallback for elements that have no particular or no meaning different from their siblings. They are typically needed as “helpers”. Using functional or generic names reduces the probability of unnecessary document or template changes.

/\* Not recommended: meaningless \*/

#yee-1901 {}

/\* Not recommended: presentational \*/

.button-green {}

.clear {}

/\* Recommended: specific \*/

#gallery {}

#login {}

.video {}

/\* Recommended: generic \*/

.aux {}

.alt {}

* 1. **ID and Class Name Delimiters**

Separate words in ID and class names by a hyphen. Do not concatenate words and abbreviations in selectors by any characters (including none at all) other than hyphens, in order to improve understanding and scan ability.

/\* Not recommended: does not separate the words “demo” and “image” \*/

.demoimage {}

/\* Not recommended: uses underscore instead of hyphen \*/

.error\_status {}

/\* Recommended \*/

#video-id {}

.ads-sample {}

1. **Formatting**

XHTML syntax must be used and all attributes must be surrounded by double quotes.

<!-- XHTML boolean attributes must still have values and self closing tags must have a closing / -->

<video autoplay="autoplay" poster="poster\_image.jpg">

<source src="foo.ogg" type="video/ogg" />

</video>

HTML5 elements should be used where appropriate reserving <div> and <span> elements for situations where there is no semantic value (such as wrapping elements to provide styling hooks).

1. **Including Meta Data**

Classes should ideally only be used as styling hooks. If you need to include additional data in the html document, for example to pass data to JavaScript, then the HTML5 data- attributes should be used.

<a class="btn" data-format="csv">Download CSV</a>

These can then be accessed easily via jQuery using the .data() method.

jQuery('.btn').data('format'); //=> "csv"

// Get the contents of all data attributes.

jQuery('.btn').data(); => {format: "csv"}

One thing to note is that the JavaScript API for datasets will convert all attribute names into camelCase. So "data-file-format" will become fileFormat.

For example:

<a class="btn" data-file-format="csv">Download CSV</a>

Will become:

jQuery('.btn').data('fileFormat'); //=> "csv"

jQuery('.btn').data(); => {fileFormat: "csv"}

1. **Targeting Internet Explorer**

Targeting lower versions of Internet Explorer (IE), those below version 9, should be handled by the stylesheets. Small fixes should be provided inline using the .ie specific class names. Larger fixes may require a separate stylesheet but try to avoid this if at all possible.

Adding IE specific classes:

<!doctype html>

<!--[if lt IE 7]> <html lang="en" class="ie ie6"> <![endif]-->

<!--[if IE 7]> <html lang="en" class="ie ie7"> <![endif]-->

<!--[if IE 8]> <html lang="en" class="ie ie8"> <![endif]-->

<!--[if gt IE 8]><!--> <html lang="en"> <!--<![endif]-->

Note: Only add lines for classes that are actually being used.

These can then be used within the CSS:

.clear:before,

.clear:after

{

content: "";

display: table;

}

.clear:after

{

clear: both;

}

.ie7 .clear

{

zoom: 1; /\* For IE 6/7 (trigger hasLayout) \*/

}

1. **HTML Validity**

Use valid HTML where possible. Use valid HTML code unless that is not possible due to otherwise unattainable performance goals regarding file size. Use tools such as the [W3C HTML validator](http://validator.w3.org/nu/) to test. Using valid HTML is a measurable baseline quality attribute that contributes to learning about technical requirements and constraints, and that ensures proper HTML usage.

<!-- Not recommended -->

<title>Test</title>

<article>This is only a test.

<!-- Recommended -->

<!DOCTYPE html>

<meta charset="utf-8">

<title>Test</title>

<article>This is only a test.</article>

1. **CSS Validity**

Use valid CSS where possible. Unless dealing with CSS validator bugs or requiring proprietary syntax, use valid CSS code. Use tools such as the [W3C CSS validator](http://jigsaw.w3.org/css-validator/) to test. Using valid CSS is a measurable baseline quality attribute that allows to spot CSS code that may not have any effect and can be removed, and that ensures proper CSS usage.

1. **Shorthand Properties**

Use shorthand properties where possible. CSS offers a variety of [shorthand](http://www.w3.org/TR/CSS21/about.html#shorthand) properties (like font) that should be used whenever possible, even in cases where only one value is explicitly set. Using shorthand properties is useful for code efficiency and understandability.

/\* Not recommended \*/

border-top-style: none;

font-family: palatino, georgia, serif;

font-size: 100%;

line-height: 1.6;

padding-bottom: 2em;

padding-left: 1em;

padding-right: 1em;

padding-top: 0;

/\* Recommended \*/

border-top: 0;

font: 100%/1.6 palatino, georgia, serif;

padding: 0 1em 2em;

1. **Prefixes**

In large projects as well as for code that gets embedded in other projects or on external sites use prefixes (as namespaces) for ID and class names. Use short, unique identifiers followed by a dash. Using namespaces helps preventing naming conflicts and can make maintenance easier, for example in search and replace operations.

.adw-help {} /\* AdWords \*/

#maia-note {} /\* Maia \*/

1. **Selector and Declaration Separation**

Separate selectors and declarations by new lines. Always start a new line for each selector and declaration.

/\* Not recommended \*/

a:focus, a:active

{

position: relative; top: 1px;

}

/\* Recommended \*/

h1,

h2,

h3

{

font-weight: normal;

line-height: 1.2;

}

1. **Section Comments**

Group sections by a CAPTIALIZED section comment. Separate sections with new lines. Do not capitalize comments within a section.

/\* HEADER \*/

#header {}

/\* Header: Navbar \*/

#header #nav

{

}

#header #nav li

{

}

/\* FOOTER \*/

#footer {}

/\* PHOTO GALLERY \*/

.photo-gallery {}

# JavaScript

1. **Naming**

The naming conventions followed for JavaScript are the same as described in the general section. The conventions that differ are described below.

* 1. **File Names**

File names should be all lower case letters and multi-word file names should be separated by underscores.

/\* Not Recommended \*/

photoGallery.js

Photo-Gallery.js

Photo\_Gallery.js

/\* Recommended \*/

photo\_gallery.js

1. **Use JavaScript files**

JavaScript programs should be stored in and delivered as .js files. JavaScript code should not be embedded in HTML files unless the code is specific to a single section of the application. Code in HTML adds significantly to page weight with no opportunity for mitigation by caching and compression.

<script src=*filename*.js> tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components. There is no need to use the language or type attributes. It is the server, not the script tag that determines the MIME type.

1. **Variables**

Always use the var keyword when declaring variables. When you fail to specify var, the variable gets placed in the global context, potentially clobbering existing values. Also, if there's no declaration, it's hard to tell in what scope a variable lives (e.g., it could be in the Document or Window just as easily as in the local scope). So always declare with var.

All variables should be declared before used. Some languages does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied [globals](http://yuiblog.com/blog/2006/06/01/global-domination/). Implied global variables should never be used.

The var statements should be the first statements in the function body.

It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order.

var currentEntry; // currently selected table entry

var level; // indentation level

var size; // size of table

JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages. Define all variables at the top of the function.

Use of global variables should be minimized. Implied global variables should never be used.

1. **Constants**

* Use NAMES\_LIKE\_THIS for constant values.
* Use @const to indicate a constant (non-overwritable) pointer (a variable or property).
* Never use the [const keyword](https://developer.mozilla.org/en/JavaScript/Reference/Statements/const) as it's not supported in Internet Explorer.

1. **Functions**

All functions should be declared before they are used. Inner functions should follow the var statements. This helps make it clear what variables are included in its scope. The body of a function is indented four spaces.

function outer(c, d) {

var e = c \* d;

function inner(a, b) {

return (e \* a) + b;

}

return inner(0, 1);

}

This convention works well with JavaScript because in JavaScript, functions and object literals can be placed anywhere that an expression is allowed. It provides the best readability with inline functions and complex structures.

function getElementsByClassName(className)

{

var results = [];

walkTheDOM(document.body, function (node) {

var a; // array of class names

var c = node.className; // the node's classname

var i; // loop counter

/\* If the node has a class name,

\* then split it into a list of simple names.

\* If any of them match the requested name,

\* then append the node to the set of results.

\*/

if (c) {

a = c.split(' ');

for (i = 0; i < a.length; i += 1)

{

if (a[i] === className)

{

results.push(node);

break;

}

}

}

});

return results;

}

**Use of global functions should be minimized.** When a function is to be invoked immediately, the entire invocation expression should be wrapped in parenthesis so that it is clear that the value being produced is the result of the function and not the function itself.

var collection = (function ()

{

var keys = [], values = [];

return

{

get: function (key)

{

var at = keys.indexOf(key);

if (at >= 0)

{

return values[at];

}

},

set: function (key, value)

{

var at = keys.indexOf(key);

if (at < 0) {

at = keys.length;

}

keys[at] = key;

values[at] = value;

},

remove: function (key)

{

var at = keys.indexOf(key);

if (at >= 0)

{

keys.splice(at, 1);

values.splice(at, 1);

}

}

};

}());

1. **Statements**
   1. **Continue Statements**

Avoid use of the continue statement. It tends to obscure the control flow of the function.

* 1. **With Statements**

The with statement [should not be used](http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/). Checkout <http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful> for more information.

1. **Semi-Colons**

Always use semi-colons. Relying on implicit insertion can cause subtle, hard to debug problems. Don't do it. You're better than that. There are a couple places where missing semicolons are particularly dangerous:

// 1.

MyClass.prototype.myMethod = function()

{

return 42;

} // No semicolon here.

(function()

{

// Some initialization code wrapped in a function to create a scope for locals.

})();

var x =

{

'i': 1,

'j': 2

} // No semicolon here.

// 2. Trying to do one thing on Internet Explorer and another on Firefox.

// I know you'd never write code like this, but throw me a bone.

[ffVersion, ieVersion][isIE]();

var THINGS\_TO\_EAT = [apples, oysters, sprayOnCheese] // No semicolon here.

// 3. conditional execution a la bash

-1 == resultOfOperation() || die();

So what happens?

1. JavaScript error - first the function returning 42 is called with the second function as a parameter, then the number 42 is "called" resulting in an error.
2. You will most likely get a 'no such property in undefined' error at runtime as it tries to call x[ffVersion, ieVersion][isIE]().
3. die is always called since the array minus 1 is NaN which is never equal to anything (not even if resultOfOperation() returns NaN) and THINGS\_TO\_EAT gets assigned the result of die().

Why?

JavaScript requires statements to end with a semicolon, except when it thinks it can safely infer their existence. In each of these examples, a function declaration or object or array literal is used inside a statement. The closing brackets are not enough to signal the end of the statement. Javascript never ends a statement if the next token is an infix or bracket operator.

This has really surprised people, so make sure your assignments end with semicolons.

Clarification: Semicolons and functions

Semicolons should be included at the end of function expressions, but not at the end of function declarations. The distinction is best illustrated with an example:

var foo = function()

{

return true;

}; // semicolon here.

function foo()

{

return true;

} // no semicolon here.

1. **Function declaration within blocks**

if (x)

{

function foo() {}

}

While most script engines support Function Declarations within blocks it is not part of ECMAScript (see [ECMA-262](http://www.ecma-international.org/publications/standards/Ecma-262.htm), clause 13 and 14). Worse implementations are inconsistent with each other and with future ECMAScript proposals. ECMAScript only allows for Function Declarations in the root statement list of a script or function. Instead use a variable initialized with a Function Expression to define a function within a block:

if (x) {

var foo = function() {};

}

1. **Some Do’s**

Here are some conventions we advise to use.

* 1. **Custom Exceptions**

Without custom exceptions, returning error information from a function that also returns a value can be tricky, not to mention inelegant. Bad solutions include passing in a reference type to hold error information or always returning Objects with a potential error member. These basically amount to a primitive exception handling hack. Feel free to use custom exceptions when appropriate.

* 1. **Use Standard Features**

These are always preferred over non-standards features. For maximum portability and compatibility, always prefer standards features over non-standards features (e.g., string.charAt(3) over string[3] and element access with DOM functions instead of using an application-specific shorthand).

# PHP

1. **Naming**

The naming conventions followed for PHP are the same as described in the general section. The conventions that differ are described below.

1. **File Naming**

PHP file naming conventions are taken from the general section above. Here we discuss PHP specific file naming conventions. However, under further sections, where the naming conventions may differ, it is stated there (as with HTML and CSS file names).

* 1. **Template Files**

Templates used in systems usually contain HTML code; however these templates are PHP templates and thus also contain PHP. Thus, template files should have the .php extension. Template files are named in all lower case letters with a dash used to separate words. Some other points to note:

* Template file names should always have the .tpl.php extension.
* Template file names should also use prefixes where possible to depict which sub-section of templates a file belongs to; for example, login.tpl.php is a form and should thus be named form-login.tpl.php.
* Template File names that contain multiple words should be separated by a hyphen.

/\* Not Recommended \*/

User-signup.php

user\_signup.tpl.php

login.tpl.php

/\* Recommended \*/

form-user-signup.tpl.php

form-login.tpl.php

* 1. **PHP Files**

PHP files should be named in all lowercase letters; words should be separated by an underscore.

index.php

user\_profile.php

some\_file.php

* 1. **Include Files**

Certain files in a PHP project would be files providing sub-functionality and in some cases, just information or some declarations. These files are usually termed include files. Include files should be named with a .inc.php extension.

settings.inc.php

constants.inc.php

categories.inc.php

user\_mgmt.inc.php

1. **PHP Code Demarcation**

Certain things are not mandatory but rather best practices; here we outline some of these.

PHP code must always be delimited by the full-form, standard PHP tags:

<?php

/\* Some code goes here \*/

?>

Short tags are never allowed. For files containing only PHP code, the closing tag must always be omitted

1. **String Literals**

For a string is literal, the double quote (“) should always be used to demarcate the string.

$a = “Example String”;

This is not mandatory for strings with no substitutions. However it becomes very useful for strings containing apostrophes. For example:

$sql = ‘SELECT id, name from \`people\` WHERE name=\'Fred\' OR name=\'Susan\'";

Can be easily and clearly written as:

$sql = "SELECT id, name from people WHERE name='Fred' OR name='Susan'";

1. **Variable Substitution**

Variable substitution is permitted using either of these forms:

$greeting = "Hello $name, welcome back!";

$greeting = "Hello {$name}, welcome back!";

This is not allowed:

$greeting = "Hello ${name}, welcome back!";

1. **Arrays**
   1. **Numerically Indexed Arrays**

* Negative numbers are not permitted as indices.
* An indexed array may start with any non-negative number, however all base indices besides 0 are discouraged.
* When declaring indexed arrays with the Array function, a trailing space must be added after each comma delimiter to improve readability:

/\* Sample array declaration \*/

$sampleArray = [array](http://www.php.net/array)(1, 2, 3, 'Zend', 'Studio');

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration:

$sampleArray = [array](http://www.php.net/array)(

    1, 2, 3, 'Zend', 'Studio',

    $a, $b, $c,

    56.44, $d, 500,

);

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

* 1. **Associative Arrays**

When declaring associative arrays with the Array construct, breaking the statement into multiple lines is encouraged. In this case, each successive line must be padded with white space such that both the keys and the values are aligned:

$sampleArray = [array](http://www.php.net/array)(

'firstKey'  => 'firstValue',

'secondKey' => 'secondValue'

);

The initial array item should begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration. It is encouraged to use a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

1. **Classes, Functions and Methods.**

For PHP classes, functions and methods, follow the coding standards specified in the general section at the start of the document.

1. **Control Statements**

For PHP control statements, follow the coding standards specified in the general section at the start of the document.

# Databases

1. **General Naming Convention**

For naming database structures (tables, fields, keys, etc.), we follow a few general naming conventions:

* Keep the name as short as possible
* Use all lower case letter
* Words are separated by an underscore.

1. **Database Users**

Each database within a server should have a different database user that only have access and privileges to that database. DO NOT use the root account to access any database or server. The user which is used to access a database should have a name that is exactly the same as the database being accessed by the user.

Password: DO NOT use the same password more than once anywhere, whether it’s on the same database server or several databases always use different passwords. Passwords should be strong including at a minimum:

* 10 characters long
* Upper case letters
* Lower case letters
* Numbers
* Special Characters: ` ~ ! @ # $ % ^ & \* ( ) \_ - + = { } [ ] \ | : ; " ' < > , . ? /

1. **Database Name**

The database name should be the same as the name of the application. For example, if you have a website named “click” or “rapidkart” or “personal\_manager”; the database names should not be:

Click\_DB

RapidKart

personalmanager

but rather, the names should be:

click

rapidkart

personal\_manager

1. **Table Name**

Table Names should always be singular. We use this naming convention because: You can think of bag containing apples like "AppleBag", it doesn't matter if contains 0, 1 or a million apples, it is always the same bag. Tables are just that, containers, the table name must describe what it contains, no how much data it contains. More information can be found at <http://cleverlogic.net/articles/should-we-use-singular-or-plural-database-table-names>.

users

User

user\_ tbl

admin\_users

adminUsers

user

admin\_user

1. **Column Name**

Column Names should, in almost all cases be singular, since in a normalized database, each cell (column value at a specific row) will only contain a singular value.

name

fullName

First\_Name

Addresses

name

full\_name

first\_name

address

In the example above, if there are many addresses for a single user, then these addresses should be moved to a separate table with all the addresses of all users, and should have foreign key references to the users table (user) pointing to which user an address is for.

In a few cases there will be need for **Plural** column names. Only in special cases (some examples given below) should plural column names be user.

* Cache Tables: Cache tables are created to improve the speed of an application and to reduce the joins and number of queries needed. For example, suppose each user in a system has its data (first name and last name)stored in the user table, 3 telephone numbers which are stored in the telephone\_number table and 3 addresses which are stored in the address table. Each time we need to show a user’s data, telephone numbers and addresses, separate queries need to be made to the above 3 tables. This extra performance burden can be removed by adding a cache\_user table which may have a structure like this:
  1. cache\_user
     1. first\_name
     2. last\_name
     3. addresses
     4. telephone\_numbers

This cache table will allow us to make only 1 query to get all of a user’s data; rather than the 3 queries we needed before.

Also, the important point here is that the addresses and telephone\_numbers fields in the cache\_user table store multiple entries, and thus can be named with plural names.