|  |
| --- |
| Code conventions |
| Agreements over the code .(end) |
|  |
| JP / Lukasz / Youssef |
|  |

# What’s coding convention ?

• • •

**Coding conventions** are a set of guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a program written in that language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc. These are guidelines for software structural quality.

# C#

## Layout Conventions

* Write only one statement per line.
* Write only one declaration per line.
* One Tab is 4 spaces
* Add at least one blank line between method definitions and property definitions.
* Use parentheses to make clauses in an expression apparent.

## Commenting Conventions

* Place the comment on a separate line, not at the end of a line of code.
* Begin comment text with an uppercase letter.
* End comment text with a period.
* Insert one space between the comment delimiter (//) and the comment text.
* Do not create formatted blocks of asterisks around comments.

## Language Guidelines

### String Data Type

* Use the + operator to concatenate short strings, as shown in the following code.
* To append strings in loops, especially when you are working with large amounts of text

### Implicitly Typed Local Variables

* Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.
* Do not use **var** when the type is not apparent from the right side of the assignment.
* Do not rely on the variable name to specify the type of the variable. It might not be correct.
* Avoid the use of **var** in place of dynamic.

### Unsigned Data Type

* In general, use int rather than unsigned types. The use of int is common throughout C#, and it is easier to interact with other libraries when you use int.

### Arrays

* Use the concise syntax when you initialize arrays on the declaration line.

### Delegates

* Use the concise syntax to create instances of a delegate type.

### && and || Operators

* To avoid exceptions and increase performance by skipping unnecessary comparisons, use && instead of & and || instead of | when you perform comparisons, as shown in the following example.

# PHP

## Whitespace

* File Encoding
* Brace Placement (Class/Control Structures/Methods)
* Blank line after <?php opening token
* Indentation (general/switch)
* Line Endings
* Space (Control Structure Brackets/Around Param Block/Method Declaration Inside Param Block)

## Naming Convention

* Class Names
* Class Constant Names
* Class Method Names
* Class Variable Names
* Constant Names
* Function Names
* Namespace Usage
* PHP Constants Casing
* PHP Keyword Casing

## Other

* Method Declare Order
* Dockblocks
* Closing Tags
* Short Tags (general/echo)
* Always use Control braces

## Castaways and Cutouts

* Class name prefixes and suffixes : In lue of using proper namespace features (older) code often falls back to using snake case and prefixes for vendors, etc. I think this will be reflected in the naming convention checks and doesn't warrent this level of detail.
* Comments and documentation blocks : Various styles could be easy to check, like //, # and /\*\*/. However, there would have to be a check to take docblocks into account /\*\* \*/, otherwise that would throw everyting of in favour of /\*\*/.
* Conventions for file names : There are various conventions on how to name a file depending on whether it contains a class, config or other forms of content but I don't think this could be reliably measured.
* Declaration of global variables and global constants : Doesn't seem worth the effort to gauge this.
* Declaration of Functions : Check whether people only declare functions as class methods or as procedural functions. Do we care? I think this is more context dependant than coding or convention style.
* Error Level : This one is also rather ambiguous as often the error level is declared outside of the project code. There could be a check for usage of error\_reporting but then integers could be used instead of constants, making the whole thing more tiresome than seems worth the effort.
* Operators and assignment (Yoda Conditions/Left Side Comparison) : This would be hard to reliably check for without doing extra code analysis (scan for the dollar sign $). Could be worthwhile to find out. Not sure.
* Inter-line alignment (Vertical Lining Up Values) : Why bother? People hardly seem to notice the difference as is.
* Line Length (hard limit/soft Limit) : I have a feeling this would rather hard to check for, since normative lines of code will be less than 80 characters long anyway.
* Usage of include\_once versus require\_once? : Doesn't seem worth the effort and can very likely be context dependant.

Usage of underscore '\_' for private/protected properties : This is a bit of PHP4 legacy. The PSR explicitly says "do not use", some others explicitly say yes. Could be interesting. Not sure.

[Examples](https://gist.github.com/potherca/6719201)

# HTML

* Use Lower Case Element Names
* Use Correct Document Type
* Close All HTML Elements
* Close Empty HTML Elements
* Use Lower Case Attribute Names
* Image Attributes
  + Always add the "alt" attribute to images. This attribute is important when the image for some reason cannot be displayed. Also, always define image width and height. It reduces flickering because the browser can reserve space for the image before loading.
* Avoid Long Code Lines
* Blank Lines and Indentation
* Setting The Viewport
  + HTML5 introduced a method to let web designers take control over the viewport, through the <meta> tag.
  + The viewport is the user's visible area of a web page. It varies with the device, and will be smaller on a mobile phone than on a computer screen.
* HTML Comments
  + Short comments should be written on one line.

# CSS

## Terminology

Concise terminology used in these standards:

selector {

property: value;

}

property: value makes a *declaration*. Selector and declarations makes a *rule*.

## Write valid CSS

All CSS code must be valid CSS3.

When using vendor prefixed properties, you can ignore CSS validation errors it generates.

## Line endings

Files should be formatted with \n as the line ending (Unix line endings), not \r\n (Windows line endings) or \r (Apple OS's).

## Encoding of CSS files

Encoding of CSS files should be set to UTF-8.

## Naming Conventions

Always use hyphens in class names. Do not use underscores or CamelCase notation.

/\* Correct \*/

.sec-nav

/\* Wrong \*/

.sec\_nav

.SecNav

## Values

Always define generic font families like sans-serif or serif.

/\* Correct \*/

font-family: "ff-din-web-1", Arial, Helvetica, sans-serif;

/\* Wrong \*/

font-family: "ff-din-web-1";

Shorten hexidecimal color values to 3 digits when possible:

background: #fff;

Do not use unit with 0.

/\* Correct \*/

.nav a {

padding: 5px 0 5px 2px;

}

/\* Wrong \*/

.nav a {

padding: 5px 0px 5px 2px;

}

Do not use default values if they are not necessary to override inherited values.

## Selectors

Selectors should be on a single line, with a space after the selector, followed by an opening brace. A selector should end with a closing brace on the next line. Next selector related the the previous one should be on the next line with one additional line space between them.

.nav li {

}

.nav a {

}

## Multiple selectors

Multiple selectors should each be on a single line, with no space after each comma.

faqs a.open,

.faqs a.close {

}

## Order of properties

Order of properties can have the following structure: box model, typography and graphic layer or order properties alphabetically.

## Comments

This comment style is used as the separator of the main sections. There are 2 empty lines before and after it:

/\* ==========================================================================

Section comment block

========================================================================== \*/

The following comment style is used as the separator of the subsections of the main sections. It has 2 empty lines before it and 1 empty line after it:

/\* Sub-section comment block

========================================================================== \*/

This comment style is used for commenting particular page elements. It has 1 empty line before it and no empty lines after it (it is immediately followed by the rules):

\* Pager \*/

.pager {

padding-bottom: 5px;

border-bottom: 1px solid #ccc;

}

Use upper case for the first letter in comments.

# JavaScript

## Variable Names

Use camelCase for identifier names (variables and functions).  
All names start with a letter.  
At the bottom of this page, you will find a wider discussion about naming rules.

## Spaces Around Operators

Always put spaces around operators ( = + - \* / ), and after commas:

var x = y + z;  
var values = ["Volvo", "Saab", "Fiat"];

## Code Indentation

Always use 4 spaces for indentation of code blocks:

function toCelsius(fahrenheit) {  
    return (5 / 9) \* (fahrenheit - 32);  
}

## Statement Rules

General rules for simple statements:

* Always end a simple statement with a semicolon.

General rules for complex (compound) statements:

* Put the opening bracket at the end of the first line.
* Use one space before the opening bracket.
* Put the closing bracket on a new line, without leading spaces.
* Do not end a complex statement with a semicolon.

## Object Rules

General rules for object definitions:

* Place the opening bracket on the same line as the object name.
* Use colon plus one space between each property and its value.
* Use quotes around string values, not around numeric values.
* Do not add a comma after the last property-value pair.
* Place the closing bracket on a new line, without leading spaces.
* Always end an object definition with a semicolon.

## Naming Conventions

Always use the same naming convention for all your code. For example:

* Variable and function names written as **camelCase**
* Global variables written in **UPPERCASE**(We don't, but it's quite common)
* Constants (like PI) written in **UPPERCASE**

# SQL

## General

### Do

* Use consistent and descriptive identifiers and names.
* Make judicious use of white space and indentation to make code easier to read.
* Store [ISO-8601](https://en.wikipedia.org/wiki/ISO_8601) compliant time and date information (YYYY-MM-DD HH:MM:SS.SSSSS).
* Try to use only standard SQL functions instead of vendor specific functions for reasons of portability.
* Keep code succinct and devoid of redundant SQL—such as unnecessary quoting or parentheses or WHEREclauses that can otherwise be derived.
* Include comments in SQL code where necessary. Use the C style opening /\* and closing \*/ where possible otherwise precede comments with -- and finish them with a new line.

### Avoid

* CamelCase—it is difficult to scan quickly.
* Descriptive prefixes or Hungarian notation such as sp\_ or tbl.
* Plurals—use the more natural collective term where possible instead. For example staff instead of employees or people instead of individuals.
* Quoted identifiers—if you must use them then stick to SQL92 double quotes for portability (you may need to configure your SQL server to support this depending on vendor).
* Object oriented design principles should not be applied to SQL or database structures.

## Naming conventions

### General

* Ensure the name is unique and does not exist as a [reserved keyword](http://www.sqlstyle.guide/#reserved-keyword-reference).
* Keep the length to a maximum of 30 bytes—in practice this is 30 characters unless you are using multi-byte character set.
* Names must begin with a letter and may not end with an underscore.
* Only use letters, numbers and underscores in names.
* Avoid the use of multiple consecutive underscores—these can be hard to read.
* Use underscores where you would naturally include a space in the name (first name becomes first\_name).
* Avoid abbreviations and if you have to use them make sure they are commonly understood.

### Tables

* Use a collective name or, less ideally, a plural form. For example (in order of preference) staff and employees.
* Do not prefix with tbl or any other such descriptive prefix or Hungarian notation.
* Never give a table the same name as one of its columns and vice versa.
* Avoid, where possible, concatenating two table names together to create the name of a relationship table. Rather than cars\_mechanics prefer services.

### Columns

* Always use the singular name.
* Where possible avoid simply using id as the primary identifier for the table.
* Do not add a column with the same name as its table and vice versa.
* Always use lowercase except where it may make sense not to such as proper nouns.

### Aliasing or correlations

* Should relate in some way to the object or expression they are aliasing.
* As a rule of thumb the correlation name should be the first letter of each word in the object’s name.
* If there is already a correlation with the same name then append a number.
* Always include the AS keyword—makes it easier to read as it is explicit.
* For computed data (SUM() or AVG()) use the name you would give it were it a column defined in the schema.

### Stored procedures

* The name must contain a verb.
* Do not prefix with sp\_ or any other such descriptive prefix or Hungarian notation.

### Uniform suffixes

The following suffixes have a universal meaning ensuring the columns can be read and understood easily from SQL code. Use the correct suffix where appropriate.

* \_id—a unique identifier such as a column that is a primary key.
* \_status—flag value or some other status of any type such as publication\_status.
* \_total—the total or sum of a collection of values.
* \_num—denotes the field contains any kind of number.
* \_name—signifies a name such as first\_name.
* \_seq—contains a contiguous sequence of values.
* \_date—denotes a column that contains the date of something.
* \_tally—a count.
* \_size—the size of something such as a file size or clothing.
* \_addr—an address for the record could be physical or intangible such as ip\_addr.

# Sources

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>

<https://gist.github.com/potherca/6719201>

<https://www.w3schools.com/html/html5_syntax.asp>

<https://github.com/xfiveco/css-coding-standards#multiple-selectors>

<https://www.w3schools.com/js/js_conventions.asp>